PPU ABI Specifications for Cell OS Lv-2

Ian Lance Taylor
Zembu Labs

© 1999, 2004 IBM Corporation
© 2003, 2004 Free Standards Group
© 2012 Sony Computer Entertainment Inc.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1; with no Invariant Sections, with no Front-Cover Texts, and with no Back-Cover Texts. A copy of the license is available from http://www.linuxbase.org/spec/refspecs/LSB_1.2.0/gLSB/gfdl.html.

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States and/or other countries: AIX, PowerPC. A full list of U.S. trademarks owned by IBM may be found at http://www.ibm.com/legal/copytrade.shtml.

Table of Contents

About This Document **4**

Related Documentation 4

Bit Notation and Other Typographic Conventions Used in This Document 5

1 Introduction **6**

2 PPU ABI Specifications **7**

Introduction 7

Low Level System Information 7

Object Files 36

Program Loading and Dynamic Linking 44

Libraries 47

APPENDIX A: Primary Differences Between PPU ABI for Cell OS Lv-2 and 64-Bit PowerPC ELF ABI **48**

About This Document

The application binary interface (ABI) for the Cell OS Lv-2 consists of the following specifications:

- Cell Broadband Engine™ ABI
- Synergistic Processor Unit (SPU) ABI
- PowerPC® Processor Unit (PPU) ABI

The PPU ABI specification is contained in this document. The other two Cell OS Lv-2 ABI specifications are contained in a separate document, the *Cell Broadband Engine™ and SPU ABI Specifications for Cell OS Lv-2*.

This document is intended for system and application programmers who develop programming language processors and other software tools for the Cell OS Lv-2.

Related Documentation

The following references were available at the specified locations on May 2, 2012. If a URL or other location is not provided in this section, the specified document can be found in the PlayStation®3 SDK Documentation package on the PlayStation®3 Developer Network website.

Cell OS Lv-2 (non ABI)

- *Cell OS Lv-2 User's Manual*

Application Binary Interfaces

- *Cell Broadband Engine™ and SPU ABI Specifications for Cell OS Lv-2*
- *64-Bit PowerPC ELF Application Binary Interface, Supplement 1.9*
  (International Business Machines Corporation, Free Standards Group)
  http://refspecs.linuxfoundation.org/ELF/ppc64/
- *System V Application Binary Interface*, edition 4.1
  (The Santa Cruz Operation, Inc., 1996)
  http://refspecs.freestandards.org/elf/gabi41.pdf
- *System V Application Binary Interface*, DRAFT; provides updates to chapters 4 and 5
  (The Santa Cruz Operation, Inc., 2001)
  http://refspecs.freestandards.org/elf/gabi4+/contents.html

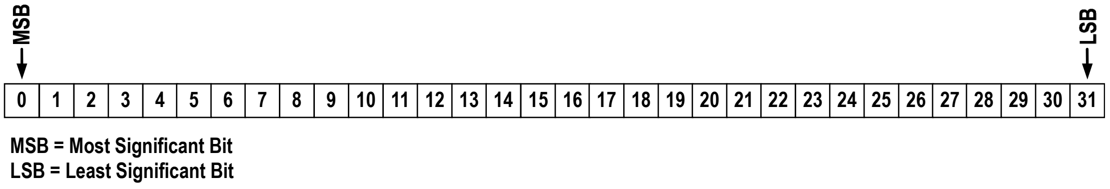
Other Information

- *PPE User's Manual*
  (included with the Cell Hardware Documentation package available on the PlayStation®3 Developer Network website)
- *PowerPC Architecture Book*, version 2.02
  (Brad Frey, International Business Machines Corporation)
  http://www.ibm.com/developerworks/eserver/library/es-archguide-v2.html
- *PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual* (PEM), version 2.07c
  (International Business Machines Corporation)
  http://www.ibm.com/chips/techlib/techlib.nsf/techdocs/C40E4C6133B31EE8872570B500791108
- *DWARF Debugging Information Format*, version 2.0.0
  (UNIX International, Inc.)
  http://refspecs.freestandards.org/dwarf/dwarf-2.0.0.pdf

Bit Notation and Other Typographic Conventions Used in This Document

Bit Notation

Standard bit notation is used throughout this document. Bits and bytes are numbered in ascending order from left to right. Thus, for a 4-byte word, bit 0 is the most significant bit and bit 31 is the least significant bit, as shown in the following figure:



Notation for bit encoding is as follows:

- Hexadecimal values are preceded by `0x`. For example: `0x0A00`.
- Binary values in sentences appear in single quotation marks. For example: '`1010`'.

Other Typographic Conventions

In addition to bit notation, the following typographic conventions are used throughout this document:

Convention	Meaning
`courier`	Indicates programming code and literals, such as processing instructions, register names, data types, events, and file names. Also indicates function and macro names. This convention is only used where it facilitates comprehension, especially in narrative descriptions.
`courier + italics`	Indicates arguments, parameters, and variables. This convention is only used where it facilitates comprehension, especially in narrative descriptions.
italics (without courier)	Indicates emphasis. Except when hyperlinked, book references are in italics. When a term is first defined, it is often in italics.
blue	Indicates a hyperlink (color printers or online only).

1 Introduction

The ABI for the Cell OS Lv-2 provides a standard interface to allow compiled objects to be linked and run on the Cell OS Lv-2, without recompilation or recoding. The purpose of these specifications is to improve application program portability.

The ABI for Cell OS Lv-2 consists of three specifications: the Cell Broadband Engine™ ABI, the SPU ABI, and the PPU ABI. The PPU specification is contained in this document. The Cell Broadband Engine™ and SPU specifications are contained in a separate document (see “About This Document”).

2 PPU ABI Specifications

Introduction

This chapter describes the ELF ABI for compiled PPU programs that run on the Cell OS Lv-2.

In general, an ELF ABI is described in two parts: a description of a generic System V ABI and a processor-specific supplement. Although the *System V Application Binary Interface* is an important reference, if the information in that document conflicts with the information here, the information in this document takes precedence.

Although the PPU ELF ABI for the Cell OS Lv-2 has much in common with the 64-bit PowerPC ELF ABI, the two specifications are not the same, nor is one a simple extension of the other.

Low Level System Information

Machine Interface

Processor Architecture

The *PowerPC Architecture Book 1: PowerPC User Instruction Set Architecture* and the *PPE User's Manual* describe the PPU instruction set and the instruction encodings and semantics of the PPU architecture. If an instruction is neither privileged nor optional, application programs can assume that all instructions that have been defined as part of the PPE architecture exist, and that they work as described.

Byte Ordering

The PPE architecture defines the following machine data types:

- 8-bit byte
- 16-bit halfword
- 32-bit word
- 64-bit doubleword
- 128-bit quadword

Byte ordering defines how the bytes that comprise halfwords, words, doublewords, and quadwords are ordered in memory. The PPU supports most significant byte (MSB) ordering. An MSB, or "big endian", ordering means that the most significant byte is located in the lowest addressed byte position in a storage unit (byte 0).

Figure 1 through Figure 4 illustrate the conventions for bit and byte numbering within various width storage units. These conventions apply to both integer data and floating-point data, where the most significant byte holds the sign and at least the start of the exponent. The following figures show byte numbers on the top and bit numbers in the lower corners.

Figure 1 Bit and Byte Numbering of Halfwords

0	1
MSB	LSB
0 7	8 15

Figure 2 Bit and Byte Numbering of Words

0	1	2	3
MSB			LSB
0 7	8 15	16 23	24 31

Figure 3 Bit and Byte Numbering of Doublewords

0	1	2	3	4	5	6	7
MSB							LSB
0 7	8 15	16 23	24 31	32 39	40 47	48 55	56 63

Figure 4 Bit and Byte Numbering of Quadwords

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
MSB															LSB
0 7	8 15	16 23	24 31	32 39	40 47	48 55	56 63	64 71	72 79	80 87	88 95	96 103	104 111	112 119	120 127

Fundamental Types

Table 1 shows the mapping of ANSI C scalar types to the PPU hardware types. The alignment column specifies the required alignment of a field of the given type within a struct. Variables can be more strictly aligned than shown in the table; however, to ensure consistent mapping, fields in a struct must use the specified alignment.

Pointers are aligned on a 4-byte boundary. Pointer size in aggregates and unions is 32 bits. In registers, pointers are always 64 bits in length. The Link Register (LR), stack pointer (SP), and TOC pointer are saved as 64 bits. See the "Stack Frame" section.

A `NULL` pointer has a value of zero.

The IBM AIX extended-precision format is not supported.

Table 1 Mapping of ANSI C Scalar Types to PPU Hardware Types

Type	ANSI C	sizeof	Alignment	PowerPC
Boolean	`_Bool`	1	byte	unsigned byte
Character	`char` `signed char`	1	byte	signed byte
	`unsigned char`	1	byte	unsigned byte
	`short` `signed short`	2	halfword	signed halfword
	`unsigned short`	2	halfword	unsigned halfword

Integral	`int`
	`signed int`
	`long`
	`signed long`
		4	word	signed word

	`enum`	4	word	unsigned/signed word
	`unsigned int` `unsigned long`	4	word	unsigned word
	`long long` `signed long long`	8	doubleword	signed doubleword
	`unsigned long long`	8	doubleword	unsigned doubleword
	`__int128_t`	16	quadword	signed quadword
	`__uint128_t`	16	quadword	unsigned quadword
Pointer	`any` * `any` (*) ()	4	word	unsigned word
Floating	`float`	4	word	single precision
	`double`	8	doubleword	double precision
	`long double`	8	doubleword	double precision

Type	ANSI C	sizeof	Alignment	PowerPC
Vector	`16*signed char`	16	quadword	vector of signed bytes
	`16*unsigned char`	16	quadword	vector of unsigned bytes
	`16*bool char`	16	quadword	vector of Boolean bytes
	`8*signed short`	16	quadword	vector of signed halfwords
	`8*unsigned short`	16	quadword	vector of unsigned halfwords
	`8*bool short`	16	quadword	vector of Boolean halfwords
	`4*signed int`	16	quadword	vector of signed words
	`4*unsigned int`	16	quadword	vector of unsigned words
	`8*pixel`	16	quadword	vector of 1/5/5/5 pixels
	`4*bool int`	16	quadword	vector of Boolean words
	`4*float`	16	quadword	vector of floats

Note: The enumerated type is signed if and only if any of its enumerators are negative.

Additional Data Types

The Cell OS Lv-2 defines additional data types to represent some fundamental data types for OS and library interfaces. These definitions are shown in Table 2.

Table 2 Definitions of Additional Data Types

Type	ANSI C	sizeof	Alignment	PowerPC
type for object size	`size_t`	4	word	unsigned int
	`ssize_t`	4	word	signed int
wide character	`wchar_t`	2	halfword	unsigned short

Aggregates and Unions

Aggregates (structures and arrays) and unions assume the alignment of their most strictly aligned component, that is, the component with the largest alignment. The size of any object, including aggregates and unions, is always a multiple of the alignment of the object.

Each member is assigned to the lowest available offset with the appropriate alignment. This might require internal padding, depending on the previous member.

In the examples shown in Figure 5 to Figure 9, the member byte offset appears in the upper-left corner of each box.

Figure 5 Structure Smaller Than a Word

```
struct {
   char c;
};
```

byte aligned, sizeof is 1

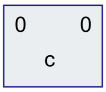


Figure 6 No Padding

```
struct {
   char c;
   char d;
   short s;
   int n;
};
```

word aligned, sizeof is 8

0 c	1 d	2 s
4 n		

Figure 7 Internal Padding

```
struct {
   char c;
   short s;
};
```

halfword aligned, sizeof is 4

0 c	1 pad	2 s

Figure 8 Internal and Tail Padding

```
struct {
   char c;
   double d;
   short s;
};
```

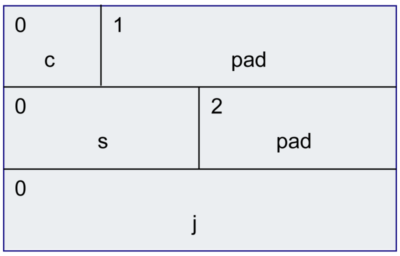
doubleword aligned, sizeof is 24

0 c	1 pad
4 pad	
8 d	
12 d	
16 s	18 pad
20 pad	

Figure 9 Union Allocation

```
union {
   char c;
   short s;
   int j;
};
```

word aligned, sizeof is 4



Bit-Fields

C struct and union definitions may have “bit-fields”, defining integral objects with a specified number of bits. The width and range of each type of bit-field is shown in Table 3. In this table, a signed range extends from - ($2^{(w-1)}$) to ($2^{(w-1)}$) - 1 and an unsigned range extends from 0 to (2^{w}) - 1.

Table 3 Bit-Field Ranges

Bit-Field Data Type	Width (w)	Range
`signed char`	1 to 8	signed
`char`		signed
`unsigned char`		unsigned
`signed short`	1 to 16	signed
`short`		signed
`unsigned short`		unsigned
`signed int`	1 to 32	signed
`int`		signed
`unsigned int`		unsigned
`enum`		unsigned/signed
`singed long`		signed
`long`		signed
`unsigned long`		unsigned
`signed long long`	1 to 64	signed
`long long`		signed
`unsigned long long`		unsigned

Note: The enum bit-field is signed if and only if the enum is signed.

“Plain” bit-fields (that is, those neither signed nor unsigned) can have either positive or negative values. Bit-fields obey the same size and alignment rules as other structure and union members, with the following additional rules:

- Bit-fields are allocated left to right (most to least significant).
- Bit-fields are limited to at most 64 bits. Adjacent bit-fields that cross the boundary of the declared type will start a new storage unit.
- The alignment of a bit-field is the same as the alignment of the base type of the bit-field. Thus, an `int` bit-field will have word alignment.
- A named, non-zero-width bit-field must share a storage unit with other structure and union members (either named non-zero-size bit-field or non-bit-field) if and only if there is sufficient space within the storage unit of the declared type.

- The data types of unnamed bit-fields do not affect the alignment of a structure or union, although an individual bit-field's member offsets obey the alignment constraints. An unnamed, zero-width bit-field must prevent any further member, bit-field or otherwise, from residing in the storage unit corresponding to the type of the zero-width bit-field.

In the examples shown in Figure 10 through Figure 16, struct and union member byte offsets are shown in the upper-left corner of each box. Bit numbers appear in the lower corners.

Figure 10 Bit Numbering

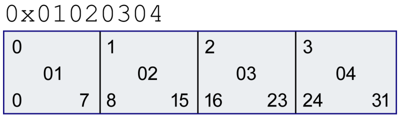


Figure 11 Bit-Field Allocation

```
struct {
   int j : 5;
   int k : 6;
   int m : 7;
};
```

word aligned, sizeof is 4

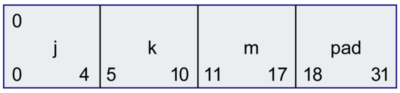


Figure 12 Boundary Alignment

```
struct {
   short s : 9;
   int j : 9;
   char c;
   short t : 9;
   short u : 9;
   char d;
};
```

word aligned, sizeof is 12

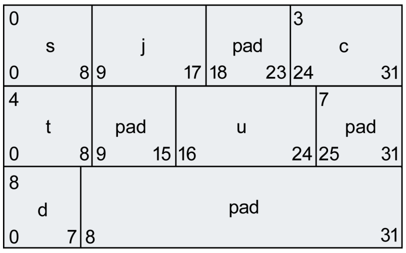


Figure 13 Doubleword Boundary Alignment

```
struct {
   long long i : 56;
   int j : 9;
};
```

doubleword aligned, sizeof is 16

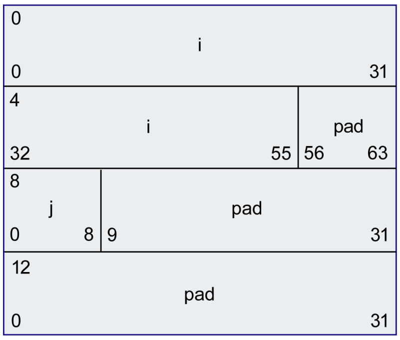


Figure 14 Storage Unit Sharing

```
struct {
   char c;
   short s : 8;
};
```

halfword aligned, sizeof is 2

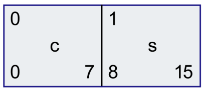


Figure 15 Union Allocation

```
union {
   char c;
   short s : 8;
};
```

halfword aligned, sizeof is 2

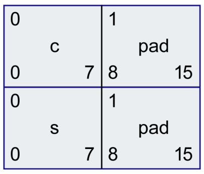
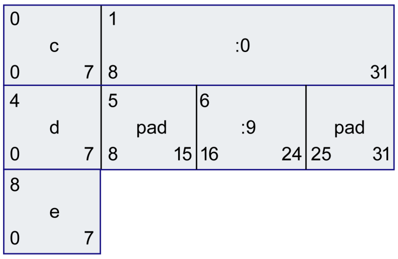


Figure 16 Unnamed Bit-Fields

```
struct {
   char c;
   int   : 0;
   char d;
   short : 9;
   char e;
};
```

byte aligned, sizeof is 9



In Figure 16, the presence of unnamed `int` and `short` fields does not affect the alignment of the structure. The named members are aligned relative to the beginning of the structure, although the named members might not be aligned in memory on suitable boundaries. For example, the `d` members in an array of these structures will not all be on an `int` (4-byte) boundary.

Function Calling Sequence

This section describes the standard function calling sequence, including stack frame layout, register usage, and parameter passing. For specific information on the implementation of C, see "Coding Examples" later in this chapter.

Standard calling-sequence requirements apply only to global functions. Local functions that are not reachable from other compilation units may use different conventions.

Registers

The PPU architecture provides 32 general purpose registers, each 64 bits wide. In addition, the architecture provides 32 floating-point registers, each 64 bits wide, and several special purpose registers. All of the integer, special purpose, and floating-point registers are global to all functions in a running program. Table 4 shows how these registers are used.

Table 4 Register Usage

Register(s)	Description
`r0`	Volatile register
`r1`	Stack frame pointer
`r2`	TOC pointer
`r3`	Volatile parameter and return value register
`r4-r10`	Volatile registers used for function parameters
`r11`	Volatile register used in calls by pointer
`r12`	Volatile register used for exception handling and .GLINK code
`r13`	Reserved for use with thread local storage
`r14-r31`	Nonvolatile registers used for local variables
`f0`	Volatile scratch register
`f1-f4`	Volatile floating-point parameter and return value registers
`f5-f13`	Volatile floating-point parameter registers
`f14-f31`	Nonvolatile registers

Register(s)	Description
LR	Volatile link register
CTR	Volatile loop counter register
XER	Volatile fixed-point exception register
FPSCR	Volatile floating-point status and control register
CR0-CR1	Volatile condition code register fields
CR2-CR4	Nonvolatile condition code register fields
CR5-CR7	Volatile condition code register fields
v0-v1	Volatile scratch registers
v2	Volatile vector parameter and return value register
v3-v13	Volatile vector parameters registers
v14-v19	Volatile scratch registers
v20-v31	Nonvolatile registers
VRSAVE	Nonvolatile 32-bit register (also see the "Process Initialization" section in this chapter)

Registers r1, r14 through r31, and f14 through f31 are nonvolatile, which means that they preserve their values across function calls. Functions that use these registers must save the value before changing it, and must restore the value before the function returns. Register r2 is technically nonvolatile but is handled specially during function calls as described in the "TOC" section below. In some cases, the calling function must restore its value after a function call.

Registers r0, r3 through r12, and f0 through f13, and the special purpose registers LR, CTR, XER, and FPSCR are volatile, which means that they are not preserved across function calls. Furthermore, registers r0, r2, r11, and r12 could be modified by cross-module calls, so a function cannot assume that the values in these registers were placed there by the calling function.

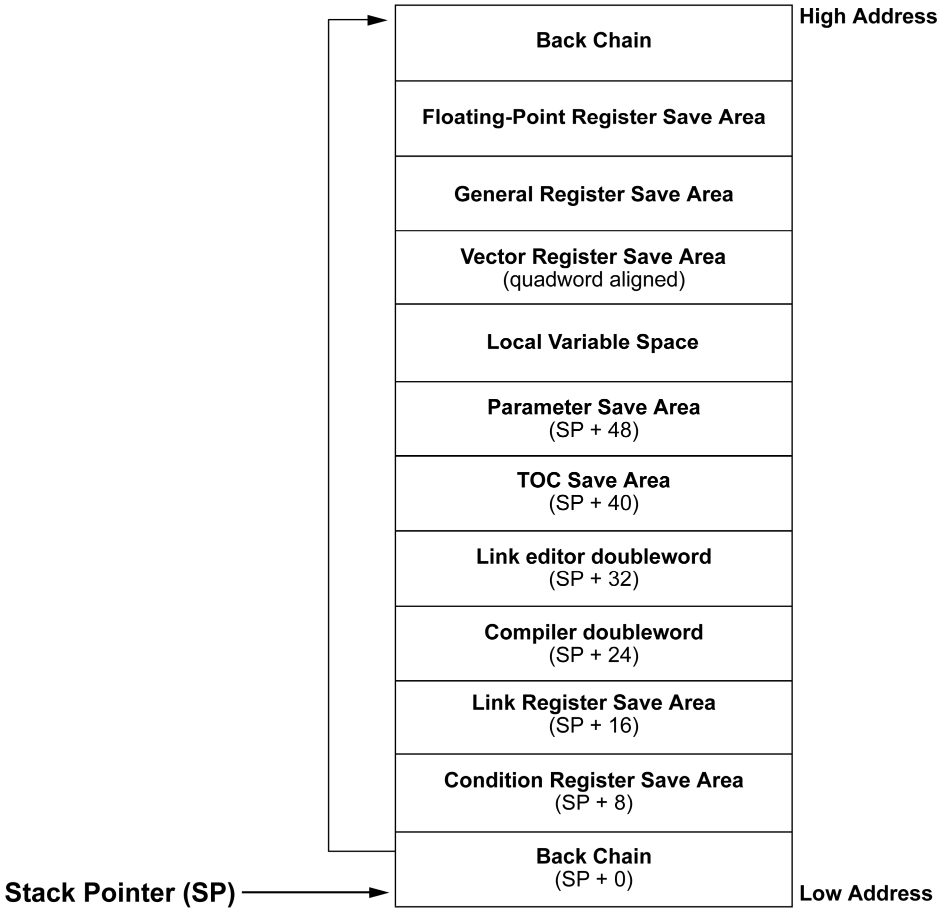
The condition code register fields CR0, CR1, and CR5 through CR7 are volatile. The condition code register fields CR2, CR3, and CR4 are nonvolatile; a function that modifies these fields must save and restore at least these fields of the condition register (CR).

The Cell OS Lv-2 does not support signals.

Stack Frame

In addition to the registers, each function may have a stack frame on the runtime stack. This stack grows downward from high addresses. The stack frame organization is shown in Figure 17. The acronym "SP" in the figure denotes the stack pointer (general purpose register r1) of the called function after it has executed code creating its stack frame.

Figure 17 Stack Frame Organization



The following requirements apply to the stack frame:

- The stack pointer must maintain quadword alignment.
- The stack pointer must point to the first word of the lowest allocated stack frame, the "Back Chain" word. The stack must grow downward, that is, toward lower addresses. The first doubleword of the stack frame must always point to the previously allocated stack frame (toward higher addresses), except for the first stack frame, which must have a Back Chain of 0 (`NULL`).
- If a stack pointer is required, it must be decremented by the called function and restored prior to its return.
- The stack pointer must be decremented and the Back Chain must be updated atomically by using one of the "Store Double Word with Update" instructions, so that the stack pointer always points to the beginning of a linked list of stack frames.
- Before a function changes the value of any nonvolatile floating-point register, `frn`, it must save the value of `frn` in the doubleword in the Floating-Point Register Save Area 8*(32-n) bytes before the Back Chain doubleword of the previous frame. The Floating-Point Register Save Area is always doubleword aligned. The size of the Floating-Point Register Save Area depends on the number of floating-point registers that must be saved. It ranges from 0 bytes to a maximum of 144 bytes (18 * 8).
- Before a function changes the value in any nonvolatile general register, `rn`, it must save the value of `rn` in the word in the General Register Save Area, 8*(32-n) bytes before the low addressed end of the Floating-Point Register Save Area. The General Register Save Area is always doubleword aligned. The size of the General Register Save Area depends on the number of general registers that must be saved. It ranges from 0 bytes to a maximum of 144 bytes (18 * 8).
- `VRSAVE` does not need to be updated in each function's prologue and epilogue because the Cell OS Lv-2 does not check the `VRSAVE` register during a context switch.

- Before a function changes the value in any nonvolatile vector register, `vrn`, it must save the value of `vrn` in the word in the Vector Register Save Area, 16*(32-n) bytes before the low addressed end of the General Register Save Area, plus alignment padding. The Vector Register Save Area is always quadword aligned. The size of the Vector Register Save Area depends on the number of vector registers that must be saved; it ranges from 0 bytes to a maximum of 192 bytes (12 * 16).
- The Local Variable Space contains any local variable storage required by the function. If vector registers are saved, the Local Variable Space will be padded so that the Vector Register Save Area is quadword aligned.
- The Parameter Save Area must be allocated by the caller. It must be quadword aligned, and must be at least 8 doublewords in length. If a function needs to pass more than 8 doublewords of arguments, the Parameter Save Area must be large enough to contain the arguments that the caller stores in it. Its contents are not preserved across function calls.
- The TOC Save Area is used by global linkage code to save the TOC pointer register.
- The link editor doubleword is reserved for use by code generated by the link editor. This ABI does not specify any usage.
- The compiler doubleword is reserved for use by the compiler.
- Before a function calls another function, it must save the contents of the Link Register in the Link Register Save Area of its caller's stack frame.
- Before a function changes the value in *any* nonvolatile field in the Condition Register, it must save the values in *all* the nonvolatile fields of the Condition Register at the time of entry to the function. These values can be saved in the Condition Register Save Area.
- The 288 bytes below the stack pointer must be available as volatile storage that is not preserved across function calls. Interrupt handlers and any other functions that might run without an explicit call must preserve this region. If a function does not need more stack space than is available in this area, it does not need to have a stack frame.

The stack frame header consists of the Back Chain word, the Condition Register Save Area, the Link Register Save Area, the compiler and link editor doublewords, and the TOC Save Area, for a total of 48 bytes.

Except for the stack frame header and any padding necessary to make the entire frame a multiple of 16 bytes in length, a function does not need to allocate space for the areas that it does not use. If a function does not call any other functions and does not require any of the other parts of the stack frame, it does not need to create a stack frame. Any padding of the frame as a whole must be within the Local Variable Space; the Parameter Save Area must immediately follow the stack frame header.

Parameter Passing

For a RISC processor, such as the PPU, it is generally more efficient to pass arguments to called functions in registers than to construct an argument list in storage or to push arguments onto a stack. Because all computations must be performed in registers, memory traffic can be eliminated if the caller can compute arguments into registers and pass them in the same registers to the called function, where the called function can then use the arguments for further computation in the same registers. Obviously, the number of registers implemented in a processor architecture limits the number of arguments that can be passed in such a way.

For the PPU, up to eight doublewords can be passed in general purpose registers. These doublewords are loaded sequentially into general purpose registers `r3` through `r10`. Up to thirteen floating-point arguments can be passed in floating-point registers `f1` through `f13`. Up to twelve vector parameters can be passed in `v2` through `v13`. If fewer (or no) arguments are passed, the unneeded registers are not loaded and will contain undefined values on entry to the called function.

The Parameter Save Area, which is located at a fixed offset of 48 bytes from the stack pointer, is reserved in each stack frame for use as an argument list. A minimum of 8 doublewords is always reserved. The size of this area must be sufficient to hold the longest argument list being passed by the function that owns the

stack frame. Although not all arguments for a particular call are located in storage, consider them to be a list in this area, with each argument occupying one or more doublewords.

If more arguments are passed than can be stored in registers, the remaining arguments are stored in the Parameter Save Area. The values passed on the stack are identical to those that have been placed in registers; thus, the stack contains register images.

For variable argument lists, a `va_list` type is used. A `va_list` is a pointer to the memory location of the next parameter. The use of a simple `va_list` type implies that variable arguments must always be in the same location regardless of type, so that they can be found at runtime. The location is defined to be general registers `r3` through `r10` for the first eight doublewords and the stack Parameter Save Area thereafter. Alignment requirements such as those for vector types may require the `va_list` pointer to be aligned before accessing a value.

The rules for parameter passing are as follows:

- Each argument is mapped to the number of doublewords in the Parameter Save Area that is required to hold the argument's value.
    - Single-precision floating-point values are mapped to the second word in a single doubleword.
    - Double-precision floating-point values are mapped to a single doubleword.
    - Simple integer types (`char`, `short`, `int`, `long`, `enum`) are mapped to a single doubleword. Values shorter than a doubleword are sign- or zero-extended as necessary.
    - Complex floating-point and complex integer types are mapped as if the argument had been specified with separate real and imaginary parts.
    - Pointers are mapped to a single doubleword.
    - Vectors are mapped to a single quadword, which is quadword aligned. This can result in skipped doublewords in the Parameter Save Area.
    - Fixed size aggregates and unions passed by value are mapped to as many doublewords of the Parameter Save Area as the value uses in memory. Aggregates and unions are aligned according to their alignment requirements. This might result in doublewords being skipped for alignment.
    - Variable size aggregates or unions are passed by reference.
    - An aggregate or union smaller than one doubleword is padded so that it appears in the least significant bits of the doubleword. All others are padded, if necessary, at their tail.
    - Other scalar values are mapped to the number of doublewords required by their size.
- If the callee has a known prototype, arguments are converted to the type of the corresponding parameter before being mapped into the Parameter Save Area. For example, if a `long` is used as an argument to a float double parameter, the value is converted to double-precision and mapped to a doubleword in the Parameter Save Area.
- Floating-point registers `f1` through `f13` are used consecutively to pass up to 13 floating-point values, complex floating point values, and one member aggregates (or aggregates within aggregates) that are passed by value that contain a floating-point value. Usually, the first 13 of all doublewords in the Parameter Save Area that map floating-point arguments will be passed in floating-point registers; the exception is arguments corresponding to the variable argument part of a callee with a prototype containing an ellipsis. A single- or double- precision value occupies one register. The corresponding complex values occupy twice as many registers.
- Usually vector registers `v2` through `v13` are used consecutively to pass up to 12 vector values; the exception is arguments corresponding to the variable argument part of a callee with a prototype containing an ellipsis. For vector arguments, an aggregate passed by value containing one vector value is treated as if the value were not wrapped in an aggregate.
- If there is no known function prototype for a callee or if the function prototype for a callee contains an ellipsis and the argument value is not part of the fixed arguments described by the prototype, floating-point and vector values are passed according to the rules for non-floating, non-vector types, described in the following paragraph. In the case of no known prototype this may result in two copies of floating and vector argument values being passed.

- General registers are used to pass some values. The first eight doublewords mapped to the Parameter Save Area correspond to the registers `r3` through `r10`. An argument that maps to this area, either fully or partially, is passed in the corresponding general registers, unless it is a floating-point or vector value that is fully described by a prototype
- All other arguments (or parts thereof) not specifically mentioned must be stored in the Parameter Save Area following the first eight doublewords. The first eight doublewords mapped to the Parameter Save Area are never stored in the Parameter Save Area by the calling function.
- If the callee takes the address of any of its parameters, values passed in registers are stored in the Parameter Save Area by the callee. If the compilation unit for the caller contains a function prototype, but the callee has a definition that is a mismatch, incorrect values might be stored.

Figure 18 Parameter Passing

```
typedef struct {
  int    a;
  double dd;
} sparm;
sparm   s, t;
int     c, d, e;
double  ff, gg, hh, ii;

x = func(c, ff, d, gg, s, hh, t, e, ii);

Parameter     Register     Offset in Parameter Save Area
c             r3           0-7     (not stored in Parameter Save Area)
ff            f1           8-15    (not stored)
d             r5           16-23   (not stored)
gg            f2           24-31   (not stored)
s             r7,r8        32-47   (not stored)
hh            f3           48-55   (not stored)
t             r10          56-71   (partially stored in Parameter
                                    Save Area)
e             (none)       72-79   (stored in Parameter Save Area)
ii            f4           80-87   (not stored)
```

If a prototype is not in scope, the following also occurs:

- The floating-point arguments `ff`, `gg`, and `hh` are passed in `r4`, `r6`, and `r9` respectively.
- The floating-point argument `ii` is stored in the Parameter Save Area.

If a prototype containing an ellipsis defines any of these floating-point arguments as being part of the variable argument part, the general registers and Parameter Save Area are used as when no prototype is in scope, and the floating-point registers are not used.

Return Values

The rules for return values are as follows:

- With the exception of pointers to float or double, functions must return float or double values in `f1`, with float values rounded to single precision. An additional exception, for floating-point complex values of size 16, is provided at the end of this subsection.
- Functions must return vector values in `v2`.
- Functions must return values of type int, long, enum, short, and char, or a pointer to any type, in `r3`, subject to the following conditions:
  - These values are unsigned or signed integers as appropriate, and are zero- or sign-extended to 64 bits if necessary.
  - Character arrays of length 8 bytes or less are returned right justified in `r3`.
  - Aggregates or unions of any length, and character arrays of length longer than 8 bytes, are returned in a storage buffer allocated by the caller. The caller will pass the address of this buffer

as a hidden first argument in r3, causing the first explicit argument to be passed in r4. This hidden argument is treated as a normal formal parameter, and corresponds to the first doubleword of the Parameter Save Area.

- Functions must return floating-point complex values of size 16 (four or eight byte complex) in f1:f2.

Passing Struct Parameters in Registers

An alternative parameter passing interface can be used for passing struct type parameters in registers. The purpose of this interface is to reduce the overhead of function calls and returns. When the d64_abi type attribute is applied to a struct type, the variables within that struct are passed in registers, if possible, in accordance with the 64-bit PowerPC Darwin ABI. This behavior only applies to struct variables that are passed by value as a function parameter and those that are returned by value as a function result. The syntax to specify a type attribute is implementation-defined.

The precise behavior of the d64_abi type attribute is as follows:

- If a parameter is a struct that is passed by value (but not unions and arrays), each field in the struct is recursively examined, and then the type of each parameter (or struct field) is examined.
- If a parameter is of type float or double, it is assigned to the next floating-point register.
- If a parameter is a vector type, it is assigned to the next vector register.

The declared return type determines whether a function result is returned in registers. If that type could be passed purely in registers as the first parameter to another function, those same registers are used to return the result.

Function Descriptors

A function descriptor is a two word data structure that contains the following values:

- The first word contains the address of the entry point of the function.
- The second word contains the TOC base address for the function.

For an externally visible function, the value of the symbol with the same name as the function is the address of the function descriptor. Symbol names with a dot (.) prefix are reserved for entry point addresses. The value of a symbol named ".FN" is the entry point of the function "FN".

The value of a function pointer in a language like C is the address of the function descriptor. For examples of using a pointer to call a function, see the section, "Coding Examples".

When the link editor processes relocatable object files in order to produce an executable or shared object, it must treat direct function calls specially, as described in the sections below.

Traceback Tables

Traceback tables are not supported; therefore, compilers are not required to emit these tables. In the absence of traceback tables, the ._eh_frame section is required for unwinding the stack, for example, for C++ exception handling.

Process Initialization

Programming language systems use this initial program state to establish a standard environment for their application programs. For example, a C program begins executing at a function named main, conventionally declared as follows:

```
extern int main (int argc, char *argv[]);
```

where argc is a non-negative argument count, and where argv is an array of argument strings, with argv[argc] == 0.

Although this section does not describe C program initialization, it describes the information that is necessary to implement a call to main or to the entry point of a program.

Registers

When a process is first entered, the contents of registers other than those listed below are unspecified. Thus, a program that requires registers to have specific values must set them explicitly during process initialization. It should not rely on the operating system to set all registers to 0. Table 5 shows the registers whose contents are specified.

Table 5 Specified Registers

Register	Description
`r1`	Contains the initial stack pointer, aligned to a quadword boundary and pointing to a word containing a `NULL` pointer.
`r2`	Contains the initial TOC pointer register value, obtained within the function descriptor pointed at by the `e_entry` field in the ELF header. For more information on function descriptors, see "Function Descriptors". For more information on the ELF Header, see "ELF Header".
`r3`	Contains `argc`, the number of arguments.
`r4`	Contains `argv`, a pointer to the array of argument pointers in the stack. The array is immediately followed by a `NULL` pointer. If there are no arguments, `r4` points to a `NULL` pointer.
`r13`	Initialized to the address of the control block of the thread local storage. This initialization is performed by the operating system when a thread is created.
`FPSCR`	Contains 0, specifying "round to nearest" mode, IEEE Mode, and the disabling of floating-point exceptions.
`VRSAVE`	Initialized to `0xffff_ffff` by the operating system when the thread is created.
`VSCR`	The NJ bit is set.
`HID1`	The `grap_mode` bit is set to `0`.

Process Stack

Every process has a stack, but the system does not define a fixed stack address. Furthermore, a program's stack address can change from one system to another, and even from one process invocation to another. Thus, the process initialization code must use the stack address in general purpose register `r1`. Data in the stack segment at addresses below the stack pointer contain undefined values.

When a process receives control, its stack holds the arguments. Argument strings appear in no specific order within the information block; the system makes no guarantees about their relative arrangement. The Back Chain word of the first stack frame contains a null pointer (`0`).

Coding Examples

This section describes example code sequences for fundamental operations such as calling functions, accessing static objects, and transferring control from one part of a program to another. Previous sections described how a program may use the machine or the operating system, and these sections specified the assumptions that a program can make about the execution environment. Unlike the previous sections, the information in this section describes how operations may be done, not how they must be done.

As before, examples use the ANSI C language. Other programming languages may use the same conventions displayed in this section, but failure to do so does not prevent a program from conforming to the ABI.

PPU code is normally position independent. That is, the code is not tied to a specific load address, and may be executed properly at various positions in virtual memory. Although it is possible to write position-dependent code on the PPU, these code examples only show position-independent code.

The examples show code fragments with various simplifications. They are intended to explain addressing modes, not to show optimal code sequences or to reproduce compiler output.

Code Model Overview

When the system creates a process image, the executable file portion of the process has fixed addresses and the system chooses shared object library virtual addresses to avoid conflicts with other segments in the process. To maximize text sharing, shared objects conventionally use position-independent code, in which instructions contain no absolute addresses. Shared object text segments can be loaded at various virtual addresses without having to change the segment images. Thus, multiple processes can share a single shared object text segment, even if the segment resides at a different virtual address in each process.

Position-independent code relies on two techniques:

- Control transfer instructions hold addresses relative to the effective address (EA) or use registers that hold the transfer address. An EA-relative branch computes its destination address in terms of the current EA, not relative to any absolute address.
- When the program requires an absolute address, it computes the desired value. Instead of embedding absolute addresses in instructions (in the text segment), during execution the compiler generates code to calculate an absolute address (in a register or in the stack or data segment).

Because the PPU instruction set provides EA-relative branch instructions and also branch instructions using registers that hold the transfer address, compilers can satisfy the first condition easily.

A "Global Offset Table", or GOT, provides information for address calculation. Position-independent executable and shared object files have a table in their data segment that holds addresses. When the system creates the memory image for an object file, the table entries are relocated to reflect the assigned absolute virtual address for an individual process. Because data segments are private for each process, the table entries can change – unlike text segments, which multiple processes share.

TOC Section

ELF processor-specific supplements normally define a GOT ("Global Offset Table") section that is used to hold addresses for position-independent code. Some ELF processor-specific supplements, including the 32-bit PowerPC Processor Supplement, define a small data section. The same register is sometimes used to address both the GOT and the small data section.

A "Table of Contents" (TOC) section contains a conventional ELF GOT. This ABI does not allow the inclusion of a small data area in the TOC section.

The TOC section is accessed through the dedicated TOC pointer register, `r2`. Access is normally made by using the register indirect with immediate index mode supported by the PPU processor, which limits a single TOC section to 65,536 bytes.

The value of the TOC pointer register is called the TOC base. The TOC base is typically the first address in the TOC plus `0x8000`, thus permitting a full 64 kilobyte TOC.

A relocatable object file must have a single TOC section and a single TOC base. However, when the link editor combines relocatable object files to form a single executable or shared object, it may create multiple TOC sections. The link editor is responsible for deciding how to associate TOC sections with object files. Normally the link editor will only create multiple TOC sections if it has more than 65,536 bytes to store in a TOC.

All link editors that support this ABI must support a single TOC section, but support for multiple TOC sections is optional.

Each shared object will have a separate TOC or TOCs.

This ABI does not actually restrict the size of a TOC section. It is permissible to use a larger TOC section, if code uses a different addressing mode to access it.

TOC Assembly Language Syntax

The ELF syntax uses `@toc` and `@got`. The syntax `SYMBOL@toc` refers to the value "`(SYMBOL - base (TOC))`", where `base (TOC)` represents the TOC base for the current object file. This expression

provides the address of the variable whose name is *SYMBOL*, as an offset from the TOC base. This assumes that the variable can be found within the TOC, and that its offset is no larger than 16 bits.

Ordinarily, the link editor will avoid having a TOC, and thus a GOT, larger than 64 kilobytes. It could avoid this by supporting multiple TOC sections, or by using some other technique. However, for flexibility, there is a syntax for 32 bit offsets to the TOC. The syntaxes `SYMBOL@toc@ha`, `SYMBOL@toc@h`, and `SYMBOL@toc@l` refer to the "high adjusted", high, and low parts of the TOC offset; the meaning of "high adjusted" is explained in the "Relocation" section of this chapter.

The syntax `SYMBOL@got` refers to the offset in the TOC at which the value of *SYMBOL* (that is, the address of the variable whose name is *SYMBOL*) is stored, assuming the offset is no larger than 16 bits.

As with the TOC, the syntaxes `SYMBOL@got@ha`, `SYMBOL@got@h`, and `SYMBOL@got@l` refer to the high adjusted, high, and low parts of the GOT offset.

The syntax `SYMBOL@got@plt` may be used to refer to the offset in the TOC of a procedure linkage table entry that is stored in the global offset table. The corresponding syntaxes `SYMBOL@got@plt@ha`, `SYMBOL@got@plt@h`, and `SYMBOL@got@plt@l` are also defined. However, these syntaxes are not used.

If *X* is a variable stored in the TOC, then `X@got` will be the offset within the TOC of a word whose value is `X@toc`.

The special symbol `.TOC.@tocbase32` is used to represent the TOC base for the current object file. The following might appear in a function descriptor definition:

```
.long .TOC.@tocbase32
```

The PowerOpen syntax is more complex. It is derived from the different representation of the TOC section in `XCOFF`.

Assembly code first uses the `.toc` pseudo-op to enter the TOC section. It then uses a label to name a particular element, and a `.tc32` pseudo-op to indicate which GOT entry it wants to name. Later in the code, the label is used with the TOC register to load the address. For example:

```
     .toc
.L1:
    .align 2
    .tc32  x[TC],x
    ...
    lwz   r3,.L1(r2)
```

This creates a GOT entry for the variable *x* and names that entry `.L1` for the remainder of the assembly. The effect is equivalent to the effect of the single ELF-style instruction that uses `@got`.

The special value `TOC[tc0]` is used to represent the TOC base for the current object file:

```
.long TOC[tc0]
```

In this ABI, relocatable object files do not contain `.got` sections holding real data. Instead, the GOT is created by the link editor based on relocations created by `@got` references. This ABI does not require the link editor to support garbage collection. This ABI does permit actual data to exist in `.toc` sections, but this data will never be referred to directly by instructions that use `@got` references. Such references always refer to the GOT that is created by the link editor when creating an executable or a shared object.

Function Prologue and Epilogue

This section describes function prologue and epilogue code. A function's prologue creates a stack frame, if necessary, and may save any nonvolatile registers it uses. A function's epilogue generally restores registers that were saved in the prologue code, restores the previous stack frame, and returns to the caller. Except for the following rules, this ABI does not mandate predetermined code sequences for function prologues and epilogues. The following rules permit reliable call chain backtracing:

- If the function uses any nonvolatile general registers, it must save them in the General Register Save Area. If the function does not require a stack frame, this may be done using negative stack offsets from the caller's stack pointer.
- If the function uses any nonvolatile floating-point registers, it must save them in the Floating-Point Register Save Area. If the function does not require a stack frame, this may be done using negative stack offsets from the caller's stack pointer.
- If the function uses any nonvolatile vector registers, it must save them in the Vector Register Save Area.
- Before a function calls any other function, it must create its own stack frame, whose size must be a multiple of 16 bytes; and the function must save the Link Register at the time of entry in the Link Register Save Area of its caller's stack frame.
- If the function uses any nonvolatile fields in the Condition Register, it may save the Condition Register in the Condition Register Save Area of the caller's stack frame.
- If a function creates a stack frame, it must use one of the "Store Double Word with Update" instructions to update the Back Chain word of the stack frame atomically with the stack pointer (`r1`):
  - For small (no larger than 32 kilobytes) stack frames, the update may be accomplished with a "Store Double Word with Update" instruction with an appropriate negative displacement.
  - For larger stack frames, the prologue must load a volatile register with the two's complement of the size of the frame (computed with `addis` and `addi` or `ori` instructions), and it must issue a "Store Double Word with Update Indexed" instruction.
- When a function deallocates its stack frame, it must do so atomically, either by loading the stack pointer (`r1`) with the value in the Back Chain field or by incrementing the stack pointer by the same amount by which it has been decremented.

In-line code may be used to save or restore nonvolatile general or floating-point registers that the function uses. However, if there are many registers to be saved or restored, it may be more efficient to call one of the system subroutines described in the sections below.

Register Saving and Restoring Functions

The register saving and restoring functions described in this section use nonstandard calling conventions that ordinarily require these functions to be statically linked into any executable or shared object modules in which they are used. Nevertheless, these functions are considered part of the ABI. In particular, the link editor is permitted to treat calls to these functions specially; for example, it can change a call to one of these functions into a call to an absolute address.

As shown in the "Stack Frame" section earlier in this chapter, the General Register Save Area is not at a fixed offset from either the caller's stack pointer or the callee's stack pointer. The Floating-Point Register Save Area starts at a fixed position from the caller's stack pointer on entry to the callee, but the position of the General Register Save Area depends on the number of floating-point registers to be saved. Thus, it is impossible to write a general register saving routine that uses fixed offsets from the stack pointer.

If a routine needs to save both general and floating-point registers, code can use `r12` as the pointer for saving and restoring the general-purpose registers; `r12` is a volatile register but does not contain input parameters. This leads to the definition of multiple register save and restore routines, each of which saves or restores *`M`* floating-point registers and *`N`* general registers.

Saving General Registers Only

For a function that saves or restores *N* general registers and no floating-point registers, save operations can be performed by using individual store/load instructions or by calling system provided routines, as shown in the following example. In this example, the number of registers being saved is *N*, and `<32-N>` is the first register number to be saved/restored. All registers from `<32-N>` through 31, inclusive, are saved/restored.

`FRAME_SIZE` is the size of the stack frame, here assumed to be less than 32 kilobytes.

```
mflr  r0                        # Move LR into r0
bl    _savegpr0_<32-N>          # Call routine to save general registers
stdu  r1,(-FRAME_SIZE)(r1)      # Create stack frame
...
(save CR if necessary)
...                             # Body of function
...
(reload CR if necessary)
...
(reload caller's SP into r1)
b     _restgpr0_<32-N>           # Restore registers and return
```

Saving General Registers and Floating-Point Registers

For a function that saves or restores *N* general registers and *M* floating-point registers, a save operation can be performed by using individual store/load instructions or by calling routines that are provided by the system, as shown in the following example:

```
mflr  r0                     # Move LR into r0
subi  r12,r1,8*M             # Set r12 to general register save area
bl    _savegpr1_<32-N>       # Call routine to save general registers
bl    _savefpr_<32-M>        # Call routine to save floating-point registers
stdu  r1,(-FRAME_SIZE)(r1)   # Create stack frame
...
(save CR if necessary)
...                          # Body of function
...
(reload CR if necessary)
...
(reload caller's SP into r1)
subi  r12,r1,8*M             # Set r12 to general register save area
bl    _restgpr1_<32-N>       # Restore general registers
b     _restfpr_<32-M>        # Restore floating-point registers and return
```

Saving Floating-Point Registers Only

For a function that saves or restores *M* floating point-registers and no general registers, a save operation can be performed by using individual store/load instructions or by calling routines that are provided by the system, as shown in the following example:

```
mflr  r0                     # Move LR into r0
bl    _savefpr_<32-M>        # Call routine to save general registers
stdu  r1,(-FRAME_SIZE)(r1)   # Create stack frame
...
(save CR if necessary)
...                          # Body of function
...
(reload CR if necessary)
...
(reload caller's SP into r1)
b     _restfpr_<32-M>        # Restore registers and return
```

Save and Restore Services

Systems must provide three sets of routines, which may be implemented as multiple entry point routines or as individual routines. They must adhere to the following rules:

- Each `_savegpr0_N` routine saves the general registers from `rN` to `r31`, inclusive.
- Each routine also saves the Link Register. When the routine is called, `r1` must point to the start of the General Register Save Area, and `r0` must contain the value of the Link Register upon function entry.
- The `_restgpr0_N` routines restore the general registers from `rN` to `r31`, and then return to the caller. When the routine is called, `r1` must point to the start of the General Register Save Area.

The remainder of this section provides sample implementations involving:

- `_savegpr0_N` and `_restgpr0_N`
- `_savegpr1_N` and `_restgpr1_N`
- `_savefpr_M` and `_restfpr_M`
- `_savevr_M` and `_restvr_M`

A sample implementation of `_savegpr0_N` and `_restgpr0_N` follows:

```
_savegpr0_14:  std  r14,-144(r1)
_savegpr0_15:  std  r15,-136(r1)
_savegpr0_16:  std  r16,-128(r1)
_savegpr0_17:  std  r17,-120(r1)
_savegpr0_18:  std  r18,-112(r1)
_savegpr0_19:  std  r19,-104(r1)
_savegpr0_20:  std  r20,-96(r1)
_savegpr0_21:  std  r21,-88(r1)
_savegpr0_22:  std  r22,-80(r1)
_savegpr0_23:  std  r23,-72(r1)
_savegpr0_24:  std  r24,-64(r1)
_savegpr0_25:  std  r25,-56(r1)
_savegpr0_26:  std  r26,-48(r1)
_savegpr0_27:  std  r27,-40(r1)
_savegpr0_28:  std  r28,-32(r1)
_savegpr0_29:  std  r29,-24(r1)
_savegpr0_30:  std  r30,-16(r1)
_savegpr0_31:  std  r31,-8(r1)
               std  r0, 16(r1)
               blr

_restgpr0_14:  ld   r14,-144(r1)
_restgpr0_15:  ld   r15,-136(r1)
_restgpr0_16:  ld   r16,-128(r1)
_restgpr0_17:  ld   r17,-120(r1)
_restgpr0_18:  ld   r18,-112(r1)
_restgpr0_19:  ld   r19,-104(r1)
_restgpr0_20:  ld   r20,-96(r1)
_restgpr0_21:  ld   r21,-88(r1)
_restgpr0_22:  ld   r22,-80(r1)
_restgpr0_23:  ld   r23,-72(r1)
_restgpr0_24:  ld   r24,-64(r1)
_restgpr0_25:  ld   r25,-56(r1)
_restgpr0_26:  ld   r26,-48(r1)
_restgpr0_27:  ld   r27,-40(r1)
_restgpr0_28:  ld   r28,-32(r1)
_restgpr0_29:  ld   r0, 16(r1)
               ld   r29,-24(r1)
               mtlr r0
               ld   r30,-16(r1)
               ld   r31,-8(r1)
```

```
                blr
_restgpr0_30:   ld   r30,-16(r1)
_restgpr0_31:   ld   r0, 16(r1)
                ld   r31,-8(r1)
                mtlr r0
                blr
```

Each `_savegpr1_N` routine saves the general registers from `rN` to `r31`, inclusive. When the routine is called, `r12` must point to the start of the General Register Save Area.

The `_restgpr1_N` routines restore the general registers from `rN` to `r31`. When the routine is called, `r12` must point to the start of the General Register Save Area.

A sample implementation of `_savegpr1_N` and `_restgpr1_N` follows:

```
_savegpr1_14:   std  r14,-144(r12)
_savegpr1_15:   std  r15,-136(r12)
_savegpr1_16:   std  r16,-128(r12)
_savegpr1_17:   std  r17,-120(r12)
_savegpr1_18:   std  r18,-112(r12)
_savegpr1_19:   std  r19,-104(r12)
_savegpr1_20:   std  r20,-96(r12)
_savegpr1_21:   std  r21,-88(r12)
_savegpr1_22:   std  r22,-80(r12)
_savegpr1_23:   std  r23,-72(r12)
_savegpr1_24:   std  r24,-64(r12)
_savegpr1_25:   std  r25,-56(r12)
_savegpr1_26:   std  r26,-48(r12)
_savegpr1_27:   std  r27,-40(r12)
_savegpr1_28:   std  r28,-32(r12)
_savegpr1_29:   std  r29,-24(r12)
_savegpr1_30:   std  r30,-16(r12)
_savegpr1_31:   std  r31,-8(r12)
                blr

_restgpr1_14:   ld   r14,-144(r12)
_restgpr1_15:   ld   r15,-136(r12)
_restgpr1_16:   ld   r16,-128(r12)
_restgpr1_17:   ld   r17,-120(r12)
_restgpr1_18:   ld   r18,-112(r12)
_restgpr1_19:   ld   r19,-104(r12)
_restgpr1_20:   ld   r20,-96(r12)
_restgpr1_21:   ld   r21,-88(r12)
_restgpr1_22:   ld   r22,-80(r12)
_restgpr1_23:   ld   r23,-72(r12)
_restgpr1_24:   ld   r24,-64(r12)
_restgpr1_25:   ld   r25,-56(r12)
_restgpr1_26:   ld   r26,-48(r12)
_restgpr1_27:   ld   r27,-40(r12)
_restgpr1_28:   ld   r28,-32(r12)
_restgpr1_29:   ld   r29,-24(r12)
_restgpr1_30:   ld   r30,-16(r12)
_restgpr1_31:   ld   r31,-8(r12)
                blr
```

Each `_savefpr_M` routine saves the floating-point registers from `fM` to `f31`, inclusive. When the routine is called, `r1` must point to the start of the Floating-Point Register Save Area, and `r0` must contain the value of the Link Register upon function entry.

The `_restfpr_M` routines restore the floating-point registers from `fM` to `f31`. When the routine is called, `r1` must point to the start of the Floating-Point Register Save Area.

A sample implementation of `_savefpr_M` and `_restfpr_M` follows:

```
_savefpr_14:  stfd f14,-144(r1)
_savefpr_15:  stfd f15,-136(r1)
_savefpr_16:  stfd f16,-128(r1)
_savefpr_17:  stfd f17,-120(r1)
_savefpr_18:  stfd f18,-112(r1)
_savefpr_19:  stfd f19,-104(r1)
_savefpr_20:  stfd f20,-96(r1)
_savefpr_21:  stfd f21,-88(r1)
_savefpr_22:  stfd f22,-80(r1)
_savefpr_23:  stfd f23,-72(r1)
_savefpr_24:  stfd f24,-64(r1)
_savefpr_25:  stfd f25,-56(r1)
_savefpr_26:  stfd f26,-48(r1)
_savefpr_27:  stfd f27,-40(r1)
_savefpr_28:  stfd f28,-32(r1)
_savefpr_29:  stfd f29,-24(r1)
_savefpr_30:  stfd f30,-16(r1)
_savefpr_31:  stfd f31,-8(r1)
              std  r0, 16(r1)
              blr

_restfpr_14:  lfd  f14,-144(r1)
_restfpr_15:  lfd  f15,-136(r1)
_restfpr_16:  lfd  f16,-128(r1)
_restfpr_17:  lfd  f17,-120(r1)
_restfpr_18:  lfd  f18,-112(r1)
_restfpr_19:  lfd  f19,-104(r1)
_restfpr_20:  lfd  f20,-96(r1)
_restfpr_21:  lfd  f21,-88(r1)
_restfpr_22:  lfd  f22,-80(r1)
_restfpr_23:  lfd  f23,-72(r1)
_restfpr_24:  lfd  f24,-64(r1)
_restfpr_25:  lfd  f25,-56(r1)
_restfpr_26:  lfd  f26,-48(r1)
_restfpr_27:  lfd  f27,-40(r1)
_restfpr_28:  lfd  f28,-32(r1)
_restfpr_29:  ld   r0, 16(r1)
              lfd  f29,-24(r1)
              mtlr r0
              lfd  f30,-16(r1)
              lfd  f31,-8(r1)
              blr
_restfpr_30:  lfd  f30,-16(r1)
_restfpr_31:  ld   r0, 16(r1)
              lfd  f31,-8(r1)
              mtlr r0
              blr
```

Each `_savevr_M` routine saves the vector registers from `vM` to `v31`, inclusive. When the routine is called, `r0` must point to the word immediately beyond the end of the Vector Register Save Area. On return, the value of `r0` is unchanged whereas `r12` may be modified.

The `_restvr_M` routines restore the vector registers from `vM` to `v31`. When the routine is called, `r0` must point to the word immediately beyond the end of the Vector Register Save Area. On return, the value of `r0` is unchanged whereas `r12` may be modified.

A sample implementation of `_savevr_M` and `_restvr_M` follows:

```
_savevr_20:   addi r12,r0,-192
              stvx v20,r12,r0
_savevr_21:   addi r12,r0,-176
              stvx v21,r12,r0
_savevr_22:   addi r12,r0,-160
              stvx v22,r12,r0
_savevr_23:   addi r12,r0,-144
              stvx v23,r12,r0
_savevr_24:   addi r12,r0,-128
              stvx v24,r12,r0
_savevr_25:   addi r12,r0,-112
              stvx v25,r12,r0
_savevr_26:   addi r12,r0,-96
              stvx v26,r12,r0
_savevr_27:   addi r12,r0,-80
              stvx v27,r12,r0
_savevr_28:   addi r12,r0,-64
              stvx v28,r12,r0
_savevr_29:   addi r12,r0,-48
              stvx v29,r12,r0
_savevr_30:   addi r12,r0,-32
              stvx v30,r12,r0
_savevr_31:   addi r12,r0,-16
              stvx v31,r12,r0
              blr

_restvr_20:   addi r12,r0,-192
              lvx  v20,r12,r0
_restvr_21:   addi r12,r0,-176
              lvx  v21,r12,r0
_restvr_22:   addi r12,r0,-160
              lvx  v22,r12,r0
_restvr_23:   addi r12,r0,-144
              lvx  v23,r12,r0
_restvr_24:   addi r12,r0,-128
              lvx  v24,r12,r0
_restvr_25:   addi r12,r0,-112
              lvx  v25,r12,r0
_restvr_26:   addi r12,r0,-96
              lvx  v26,r12,r0
_restvr_27:   addi r12,r0,-80
              lvx  v27,r12,r0
_restvr_28:   addi r12,r0,-64
              lvx  v28,r12,r0
_restvr_29:   addi r12,r0,-48
              lvx  v29,r12,r0
_restvr_30:   addi r12,r0,-32
              lvx  v30,r12,r0
_restvr_31:   addi r12,r0,-16
              lvx  v31,r12,r0
              blr
```

Data Objects

This section describes objects with static storage duration. It excludes stack-resident objects because programs always compute their virtual addresses relative to the stack or frame pointers.

In the PPE architecture, only load and store instructions access memory. Because PPU instructions cannot directly hold either 64-bit addresses or 32-bit addresses, a program normally computes an address into a register and accesses memory through the register.

It is possible to build addresses using absolute code that puts symbol addresses into instructions. However, to support dynamic libraries, by default, PPU code normally loads an address from a memory location in the TOC section. Combining the TOC offset of the symbol with the TOC address in register `r2` gives the absolute address of the TOC entry holding the desired address.

The following examples show sample assembly language equivalents to C language code. The `@got` syntax is explained earlier in this chapter, in "TOC Assembly Language Syntax".

Example of Load and Store; variables are not in TOC:

```
C                              Assembly

extern int src;
extern int dst;
extern int *ptr;


dst = src;
                              lwz  r6,src@got(r2)
                              lwz  r7,dst@got(r2)
                              lwz  r0,0(r6)
                              stw  r0,0(r7)

ptr = &dst;
                              lwz  r0,dst@got(r2)
                              lwz  r7,ptr@got(r2)
                              stw  r0,0(r7)

*ptr = src;
                              lwz  r6,src@got(r2)
                              lwz  r7,ptr@got(r2)
                              lwz  r0,0(r6)
                              lwz  r7,0(r7)
                              stw  r0,0(r7)
```

Function Calls

In a relocatable object file, a direct function call should be made to the function entry point, which is a symbol beginning with a dot (.).

When the link editor is creating an executable or shared object, and it sees a function call followed by a No-Operation (NOP) instruction, it determines whether the caller and the callee share the same TOC. If they do, the NOP instruction is not changed. If they do not, the link editor constructs a linkage function. The linkage function loads the TOC register with the callee TOC and branches to the callee entry point. The link editor modifies the `bl` instruction to branch to the linkage function, and changes the NOP instruction to:

```
ld   r2,40(r1)
```

This will reload the TOC register from the TOC Save Area after the callee returns.

The link editor will not accept the following instructions as valid NOP instructions:

```
cror 15,15,15
cror 31,31,31
```

Programs use the `bl` instruction to make direct function calls. The bl instruction must be followed by a NOP instruction. The NOP instruction may be:

```
ori  r0,r0,0
```

A `bl` instruction has a self-relative branch displacement that can reach 32 megabytes in either direction. Thus, using a `bl` instruction to make a call within an executable or shared object file limits the size of the executable or shared object file text segment, unless the link editor places an additional branch in the executable to extend the effective length of the `bl` instruction.

If the callee is in a different shared object, a similar procedure of linkage code and a modified NOP instruction is used. In this case, the dynamic linker must complete the link by inserting the function descriptor at runtime.

The following is an example of the assembly code generated for a call to a non-local function or for a function with no prototype:

```
C                                 Assembly

extern void func (void);
func ();
                                  bl   .func
                                  ori  r0,r0,0
```

Because most of the code sequence used for a call through a pointer is the same regardless of the function pointer that is being used, it is also possible to call a library function to execute these instructions rather than using the standard calling convention. For efficiency, this approach requires that the function be linked in directly rather than coming from a shared library. A function named `._ptrgl12` is used for this purpose, and it is recommended that this name and calling convention be used.

The following code shows an example of the linkage code created by the link editor. Notice that `func@got@plt` contains the address of the procedure linkage entry for `func`, which is a function descriptor. The function descriptor holds the addresses of the function entry point and the function TOC base.

```
<linkage_for_func>:
    lwz    r12,func@got@plt(r2)
    std    r2,40(r1)
    lwz    r0,0(r12)
    lwz    r2,4(r12)
    mtctr  r0
    bctr
```

The value of a function pointer is the address of the function descriptor, not the address of the function entry point itself.

```
C                                 Assembly

extern void func (void);
extern void (*ptr) (void);
ptr = func;
                                  lwz    r6,func@got(r2)
                                  lwz    r7,ptr@got(r2)
                                  stw    r6,0(r7)

(*ptr) ();
                                  lwz    r6,ptr@got(r2)
                                  lwz    r6,0(r6)
                                  lwz    r0,0(r6)
```

```
                                  std     r2,40(r1)
                                  mtctr   r0
                                  lwz     r2,4(r6)
                                  bctrl
                                  ld      r2,40(r1)
```

Branching

Programs use branch instructions to control their execution flow. As defined by the PPU architecture, branch instructions hold a self-relative value with a 64-megabyte range, allowing a jump to locations up to 32 megabytes away in either direction.

```
C                                 Assembly
label:
                                  .L01:
    ...
    goto label
                                      b .L01
```

C switch statements provide multi-way selection. When the case labels of a switch statement satisfy grouping constraints, the compiler implements the selection with an address table. The following example uses several simplifying conventions to hide irrelevant details. In this example:

- The selection expression resides in `r12`, and is of type `int`.
- The case label constants begin at zero.
- The case labels, the default, and the address table use the assembly names `.Lcasei`, `.Ldef`, and `.Ltab`, respectively.

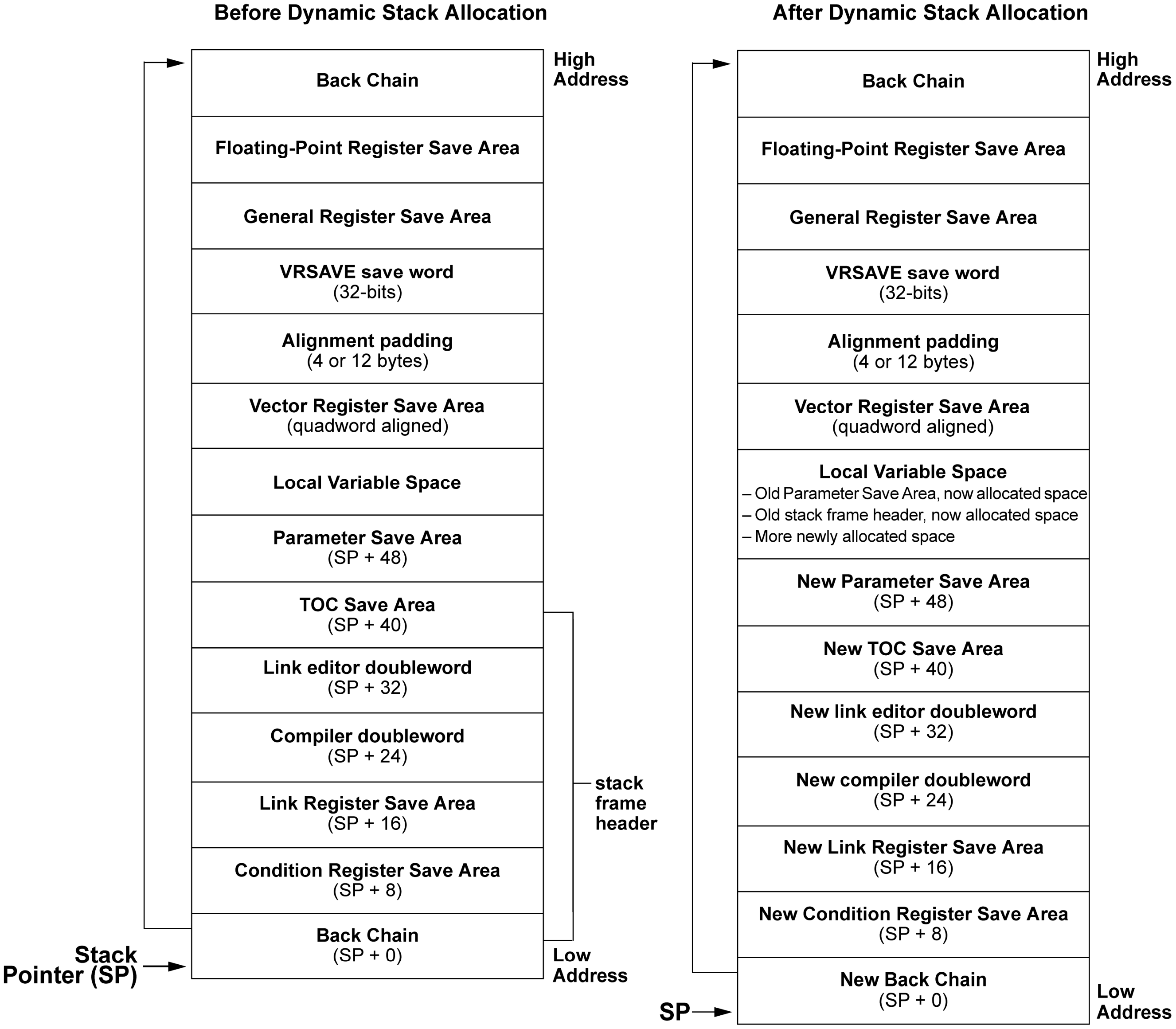
```
C                              Assembly

switch (j)
  {
  case 0:
    ...
  case 1:
    ...
  case 3:
    ...
  default:
    ...
  }
                                   cmplwi  r12,4
                                   bge     .Ldef
                                   bl      .L1
                               .L1:
                                   slwi    r12,2
                                   mflr    r11
                                   addi    r12,r12,.Ltab-.L1
                                   add     r0,r12,r11
                                   mtctr   r0
                                   bctr
                               .Ltab:
                                   b       .Lcase0
                                   b       .Lcase1
                                   b       .Ldef
                                   b       .Lcase3
```

Dynamic Stack Space Allocation

To support variable-length arrays, ISO C99 requires support for dynamic stack allocation within a stack frame. The mechanism for allocating dynamic space is embedded completely within a function and does not affect the standard calling sequence. Thus, functions that need dynamic stack frame sizes can call functions that do not need them, and the reverse is also true.

Figure 19 shows the stack frame before and after dynamic stack allocation.

Figure 19 Stack Frame Before and After Dynamic Stack Allocation



The Local Variable Space is used for storage of function data, such as local variables, whose sizes are known to the compiler. This area is allocated upon function entry and does not change in size or position during the function's activation.

The Parameter Save Area is reserved for arguments passed in calls to other functions; see "Parameter Passing" for more information. Its size is also known to the compiler and, upon function entry, may be allocated along with the fixed frame area. However, the standard calling sequence requires that the Parameter Save Area begin at a fixed offset (48) from the stack pointer, so this area must move when dynamic stack allocation occurs.

The stack frame header must also be at a fixed offset (0) from the stack pointer, so this area must also move when dynamic stack allocation occurs.

Data in the Parameter Save Area is naturally addressed at constant offsets from the stack pointer. However, in the presence of dynamic stack allocation, the offsets from the stack pointer to the data in the local variables area are not constant. To provide addressability, a frame pointer is used to locate the local variables area consistently throughout the function's activation.

Dynamic stack allocation is accomplished by "opening" the stack directly above the Parameter Save Area. The following steps describe this procedure in more detail:

(1) After a new stack frame is acquired but before the first dynamic space allocation, a new register, the frame pointer, is set to the value of the stack pointer. The frame pointer is used for references to the function's local, non-static variables.

(2) The amount of dynamic space to be allocated is rounded up to a multiple of 16 bytes, so that quadword stack alignment is maintained.

(3) The stack pointer is decreased by the rounded byte count, and the address of the previous stack frame (the Back Chain) is stored at the word addressed by the new stack pointer. This must be accomplished atomically by using `stdu rS,-length(r1)` if the length is less than 32768 bytes, or by using `stdux rS,r1,rspace`, where `rS` is the contents of the Back Chain word and `rspace` contains the (negative) rounded number of bytes to be allocated.

It is only strictly necessary to copy the Back Chain. The information in the Parameter Save Area is recreated for each function call. The information in the stack frame header, other than the Back Chain, is only used by a called function. In some cases, a compiler may need to copy the TOC Save Area as well, depending on how it generates linkage code.

This procedure can be repeated as many times as desired within a single function activation. When it is time to return, the stack pointer is set to the value of the Back Chain, thus removing all dynamically allocated stack space along with the rest of the stack frame. Naturally, a program must not reference the dynamically allocated stack area after it has been freed.

DWARF Definition

DWARF Release Number

This section defines the Debug With Arbitrary Record Format (DWARF) debugging format for the PPU. The PPU ABI does not define a debug format; however, all systems that do implement DWARF must use the following definitions.

DWARF is a specification developed for symbolic, source-level debugging. The debugging information format does not favor the design of any compiler or debugger. For more information on DWARF, see *DWARF Debugging Information Format*; a link is provided in the "Related Documentation" section of this document.

The DWARF definition requires some machine-specific definitions. The register number mapping must be specified for the 64-bit PowerPC registers. In addition, the DWARF (version 2) specification requires processor-specific address class codes to be defined.

DWARF Register Number Mapping

Table 6 shows the register number mapping for the PPU. It is the same mapping used for the 64-bit PowerPC processor family. Notice that for all special purpose registers (SPRs), the number is simply 100 plus the SPR register number, as defined in the 64-bit PowerPC architecture.

Table 6 DWARF Register Mapping for PPU

Register Name	Number(s)	Abbreviation(s)
General Register 0–31	0–31	`r0-r31`
Floating Register 0–31	32–63	`f0-f31`
Condition Register	64	`CR`
Floating-Point Status and Control Register	65	`FPSCR`

Register Name	Number(s)	Abbreviation(s)
MQ Register*	100	MQ or SPR0
Fixed-Point Exception Register	101	XER or SPR1
Real Time Clock Upper Register*	104	RTCU or SPR4
Real Time Clock Lower Register*	105	RTCL or SPR5
Link Register	108	LR or SPR8
Count Register	109	CTR or SPR9

* Register names followed with an asterisk are MPC601 chip-specific and are not part of the generic 64-bit PowerPC chip architecture.

For kernel debuggers, the mapping for all privileged registers is also defined. See Table 7.

Table 7 DWARF Privileged Register Mapping for Kernel Debuggers

Register Name	Number(s)	Abbreviation(s)
Machine State Register	66	MSR
Segment Register 0–15	70-85	SR0-SR15
Data Storage Interrupt Status Register	118	DSISR or SPR18
Data Address Register	119	DAR or SPR19
Decrementer	122	DEC or SPR22
Storage Description Register 1	125	SDR1 or SPR25
Machine Status Save/Restore Register 0	126	SRR0 or SPR26
Machine Status Save/Restore Register 1	127	SRR1 or SPR27
Vector Save/Restore Register	356	VRSAVE or SPR256
software use, Special Purpose Register 0	372	SPRG0 or SPR272
software use, Special Purpose Register 1	373	SPRG1 or SPR273
software use, Special Purpose Register 2	374	SPRG2 or SPR274
software use, Special Purpose Register 3	375	SPRG3 or SPR275
Address Space Register	380	ASR or SPR280
External Access Register	382	EAR or SPR282
Time Base	384	TB or SPR284
Time Base Upper	385	TBU or SPR285
Processor Version Register	387	PVR or SPR287
Instruction BAT Register 0 Upper	628	IBAT0U or SPR528
Instruction BAT Register 0 Lower	629	IBAT0L or SPR529
Instruction BAT Register 1 Upper	630	IBAT1U or SPR530
Instruction BAT Register 1 Lower	631	IBAT1L or SPR531
Instruction BAT Register 2 Upper	632	IBAT2U or SPR532
Instruction BAT Register 2 Lower	633	IBAT2L or SPR533
Instruction BAT Register 3 Upper	634	IBAT3U or SPR534
Instruction BAT Register 3 Lower	635	IBAT3L or SPR535
Data BAT Register 0 Upper	636	DBAT0U or SPR536
Data BAT Register 0 Lower	637	DBAT0L or SPR537
Data BAT Register 1 Upper	638	DBAT1U or SPR538
Data BAT Register 1 Lower	639	DBAT1L or SPR539
Data BAT Register 2 Upper	640	DBAT2U or SPR540
Data BAT Register 2 Lower	641	DBAT2L or SPR541
Data BAT Register 3 Upper	642	DBAT3U or SPR542
Data BAT Register 3 Lower	643	DBAT3L or SPR543
Hardware Implementation Register 0*	1108	HID0 or SPR1008
Hardware Implementation Register 1*	1109	HID1 or SPR1009
Hardware Implementation Register 2*	1110	HID2 or IABR or SPR1010
Hardware Implementation Register 5*	1113	HID5 or DABR or SPR1013
Hardware Implementation Register 15*	1123	HID15 or PIR or SPR1023
Vector Registers 0–31	1124-1155	vr0-vr31

* Register names followed with an asterisk are MPC601 chip-specific and are not part of the generic 64-bit PowerPC chip architecture.

Table 8 shows address class codes for the 64-bit PowerPC processor family, which includes PPU.

Table 8 DWARF Address Class Codes for 64-Bit PowerPC Processor Family

Code	Value	Meaning
ADDR_none	0	No class specified.

Object Files

This section defines information related to the ELF header and special sections in PPU object files.

ELF Header

The machine-specific information that applies to PPU objects is shown in Table 9.

Table 9 PPU ELF Header Fields

Field	Value	Comments
e_ident[EI_CLASS]	ELFCLASS64	Specifies that PPU objects are 64-bit objects.
e_ident[EI_OSABI]	ELFOSABI_CELLOSLV2	102 (0x66) is tentatively assigned.
e_ident[EI_DATA]	ELFDATA2MSB	–
e_ident[EI_ABIVERSION]	0	A version is not defined.
e_machine	EM_PPC64	Specifies that the PPU supports a 64-bit PowerPC architecture.

The ELF header's *e_flags* member holds bit flags associated with the PPU object file. This ABI defines the flags shown in Table 10.

Table 10 PPU ELF Flags

Flag	Value	Description
EF_PPC64_REL24	0x01000000	Indicates that the R_PPC64_REL24 relocation type is left unresolved.
EF_SNC_DEDUPLICATABLE	0x02000000	Reserved by SNC for de-duplication.

Although the address range is limited to 32 bits, object and executable files are saved in ELFCLASS64 format.

Processor identification resides in the ELF header's *e_machine* member, and must have the value 21, defined as the name EM_PPC64.

The *e_entry* field in the ELF header holds the address of a function descriptor. This function descriptor supplies both the address of the function entry point and the initial value of the TOC pointer register.

Special Sections

Various sections hold program and control information. Table 11 shows the special sections used by the system; the table also shows their types, attributes, and usage. Tools that support this ABI are not required to use these sections exactly as defined in Table 11, and indeed are not required to use them at all. The true use of a section is defined by the relocation information and by the code that refers to it. However, if a tool uses these sections, each section must use the types and attributes specified in Table 11.

Table 11 Descriptions of Special Sections

Name	Type	Attributes	Description
`.glink`	`SHT_PROGBITS`	`SHF_ALLOC + SHF_EXECINSTR`	This section is not used.
`.got`	`SHT_PROGBITS`	`SHF_ALLOC + SHF_WRITE`	May be used to hold the Global Offset Table, or GOT. For more information, see "TOC Section", "Coding Examples" and "Global Offset Table".
`.toc`	`SHT_PROGBITS`	`SHF_ALLOC + SHF_WRITE`	May be used to hold the initialized Table of Contents, or TOC. For more information, see "TOC", below, and "TOC Section", "Coding Examples" and "Global Offset Table".
`.tocbss`	`SHT_NOBITS`	`SHF_ALLOC + SHF_WRITE`	This section is not used.
`.plt`	`SHT_NOBITS`	`SHF_ALLOC + SHF_WRITE`	This section is not used.
`.PpuGUID`	`SHT_PROGBITS`	`SHF_ALLOC`	This section contains 32-byte PPU GUID. A symbol "`__PPU_GUID`" is defined at the start of this section. PPU GUID contains the following five items: – PPU GUID magic (5 bytes): The contents of each of the bytes are `'\177','G','U','I','D'`. – PPU GUID revision (1 byte): The valid revision is 1. – Reserved area 1 (2 bytes): This field is filled with zeroes. – Hash value (20 bytes): A hash value created from an executable. – Reserved area 2 (4 bytes): This field is filled with zeroes.

Note: The `.plt` section is of type `SHT_NOBITS`, rather than `SHT_PROGBITS` as on most processors.

TOC

This section describes a typical layout of the TOC in an executable file or shared object. Particular tools do not need to follow the layout specified in this section.

The Table of Contents, or TOC, is part of the data segment of an executable program. The TOC typically contains data items within the `.got`, `.toc` and `.tocbss` sections, which can be addressed with 16-bit signed offsets from the TOC base. The TOC base is typically the first address in the TOC plus `0x8000`, thus permitting a full 64 kilobyte TOC. The `.got` section is typically created by the link editor based on `@got` relocations. The `.toc` and `.tocbss` sections are typically included from relocatable object files referenced during the link.

The TOC may straddle the boundary between initialized and uninitialized data in the data segment. Some of the sections in the data segment may be empty; the usual order of these sections is as follows:

(1) `.data`

(2) `.got`

(3) `.toc`

(4) `.tocbss`

(5) `.plt`

The link editor may create multiple TOC sections, as specified in the "TOC Section" in "Coding Examples" earlier in this chapter. In such a case, the `.got` and `.toc` sections will be repeated as necessary, and possibly renamed to preserve unique section names. Any occurrence of `.tocbss` in a TOC section other than the last one will be converted into a `.toc` section initialized to contain zero bytes.

Compilers may generate "short-form" single-instruction references for all data items that are in the TOC section for the object file being compiled. Such references are relative to the TOC pointer register, `r2`, which always holds the base of the TOC section for the object file.

In a shared object, only data items with local (non-global) scope may be addressed through the TOC pointer. Global data items must be addressed via the GOT, even if they appear in a `.toc` or `.tocbss` section.

A compiler that places some data items in the TOC must provide an option to avoid doing so in a particular compilation.

Symbol Table

If an executable file contains a reference to a function defined in one of its associated shared objects, the symbol table section for the file will contain an entry for that symbol. The `st_shndx` member of that symbol table entry contains `SHN_UNDEF`. This informs the dynamic linker that the symbol definition for that function is not contained in the executable file itself. The `st_value` member for that symbol entry contains zero, which means no procedure linkage table entry is associated to it.

Relocation

Relocation Types

Relocation entries describe how to change the instruction and data relocation fields. Relocation is performed on halfwords, words, and doublewords. In all cases, the `r_offset` value designates the offset or virtual address of the first byte of the affected storage unit. The relocation type specifies which bits to change and how to calculate their values. The PPU uses only the `Elf32_Rela` relocation entries with explicit addends. For the relocation entries, the `r_addend` member serves as the relocation addend. In all cases, the offset, the addend, and the computed result use the byte order specified in the ELF header.

The calculations shown in Table 12 assume that the actions are transforming a relocatable file into either an executable or a shared object file; relocations applied to executable or shared object files are similar and accomplish the same result. Conceptually, the link editor merges one or more relocatable files to form the output. It first determines how to combine and locate the input files. Next, it updates the symbol values, and then it performs the necessary relocations. Figure 20 and its accompanying table show the relocation fields and their descriptions. In the figure, bit numbers appear in the lower portion of the boxes; big-endian byte numbers appear in the upper portion.

Figure 20 Relocation Fields

```
+-------+-------+-------+-------+-------+-------+-------+-------+
|0      1       2       3       4       5       6       7       |
|                           doubleword64                        |
|0                                                            63|
+---------------------------------------------------------------+

+-------+-------+-------+--+----+
|0      1       2       3       |
|            word32             |
|0                            31|
+-------------------------------+

+-------+-------+-------+--+----+
|0      1       2       3 |     |
|            word30        |     |
|0                      29|3031|
+--------------------------+----+

+----+--+-------+-------+--+----+
|0   |  1       2       3 |     |
|    |       low24        |     |
|0  5|6                  29|3031|
+--------------------------+----+

+-------+-------+-------+--+----+
|0      1|  |  |2       3 |     |
|        |  |  | low14    |     |
|0       |10|15|16      29|3031|
+--------------------------+----+

+-------+----+--+
|0      1       |
|    half16     |
|0            15|
+---------------+

+-------+-------+--+
|0      1       |  |
|    half16ds   |  |
|0            13|15|
+---------------+--+
```

Type	Description
doubleword64	Specifies a 64-bit field occupying 8 bytes, the alignment of which is 8 bytes unless otherwise specified.
word32	Specifies a 32-bit field occupying 4 bytes, the alignment of which is 4 bytes unless otherwise specified.
word30	Specifies a 30-bit field contained within bits 0–29 of a word with 4-byte alignment. The two least significant bits of the word are unchanged.
low24	Specifies a 24-bit field contained within a word with 4-byte alignment. The six most significant and the two least significant bits of the word are ignored and unchanged (for example, “Branch” instruction).
low14	Specifies a 14-bit field contained within a word with 4-byte alignment, comprising a conditional branch instruction. The 14-bit relative displacement in bits 16–29, and possibly the “branch prediction bit” (bit 10), are altered; all other bits remain unchanged.
half16	Specifies a 16-bit field occupying 2 bytes with 2-byte alignment (for example, the immediate field of an “Add Immediate” instruction).
half16ds	Similar to **half16**, but effectively only 14 bits because the two least significant bits must be zero, and are not actually part of the field. (Used, for example, by the `ldu` instruction.)

Some relocations use high adjusted values. These are the most significant bits, adjusted so that adding the low 16 bits will perform the correct calculation of the address accounting for signed arithmetic. The purpose is to support using the low 16 bits as a signed offset when loading the value. For example, a value could be loaded from an absolute 32-bit address SYM as follows:

```
li    r3,0
oris  r3,r3,SYM@ha
ld    r4,SYM@l(r3)
```

The code assumes that SYM is less than 0xffff_8000. The adjusted forms mean that this will work correctly even if SYM@l is negative when interpreted as a signed 16 bit number. Compare this to building the same 32-bit address using ori, in which case the adjusted forms are not used:

```
li    r3,0
oris  r3,r3,SYM@h
ori   r3,r3,SYM@l
ld    r4,0(r3)
```

These code samples are not meant to encourage people to write code that constructs absolute 32-bit addresses in this manner. It is normally better to use position-independent code. However, this ABI does make this usage possible.

Table 12 shows relocation types. (See the notes following Table 12 for an explanation of the notational conventions used in the table.)

Table 12 Relocation Types

Name[1]	Value[2]	Field[3]	Calculation[4]
R_PPC64_NONE	0	none	none
R_PPC64_ADDR32	1	word32*	S + A
R_PPC64_ADDR24	2	low24*	(S + A) >> 2
R_PPC64_ADDR16	3	half16*	S + A
R_PPC64_ADDR16_LO	4	half16	#lo(S + A)
R_PPC64_ADDR16_HI	5	half16	#hi(S + A)
R_PPC64_ADDR16_HA	6	half16	#ha(S + A)
R_PPC64_ADDR14	7	low14*	(S + A) >> 2
R_PPC64_ADDR14_BRTAKEN	8	low14*	(S + A) >> 2
R_PPC64_ADDR14_BRNTAKEN	9	low14*	(S + A) >> 2
R_PPC64_REL24	10	low24*	(S + A - P) >> 2
R_PPC64_REL14	11	low14*	(S + A - P) >> 2
R_PPC64_REL14_BRTAKEN	12	low14*	(S + A - P) >> 2
R_PPC64_REL14_BRNTAKEN	13	low14*	(S + A - P) >> 2
R_PPC64_GOT16	14	half16*	G
R_PPC64_GOT16_LO	15	half16	#lo(G)
R_PPC64_GOT16_HI	16	half16	#hi(G)
R_PPC64_GOT16_HA	17	half16	#ha(G)
R_PPC64_COPY	19	none	none
R_PPC64_GLOB_DAT	20	doubleword64	S + A
R_PPC64_JMP_SLOT	21	none	see below
R_PPC64_RELATIVE	22	doubleword64	B + A
R_PPC64_UADDR32	24	word32*	S + A
R_PPC64_UADDR16	25	half16*	S + A
R_PPC64_REL32	26	word32*	S + A - P
R_PPC64_PLT32	27	word32*	L
R_PPC64_PLTREL32	28	word32*	L - P
R_PPC64_PLT16_LO	29	half16	#lo(L)
R_PPC64_PLT16_HI	30	half16	#hi(L)
R_PPC64_PLT16_HA	31	half16	#ha(L)

Name[1]	Value[2]	Field[3]	Calculation[4]
R_PPC64_SECTOFF	33	half16*	R + A
R_PPC64_SECTOFF_LO	34	half16	#lo(R + A)
R_PPC64_SECTOFF_HI	35	half16	#hi(R + A)
R_PPC64_SECTOFF_HA	36	half16	#ha(R + A)
R_PPC64_ADDR30	37	word30	(S + A - P) >> 2
R_PPC64_ADDR64	38	doubleword64	S + A
R_PPC64_ADDR16_HIGHER	39	half16	#higher(S + A)
R_PPC64_ADDR16_HIGHERA	40	half16	#highera(S + A)
R_PPC64_ADDR16_HIGHEST	41	half16	#highest(S + A)
R_PPC64_ADDR16_HIGHESTA	42	half16	#highesta(S + A)
R_PPC64_UADDR64	43	doubleword64	S + A
R_PPC64_REL64	44	doubleword64	S + A - P
R_PPC64_PLT64	45	doubleword64	L
R_PPC64_PLTREL64	46	doubleword64	L - P
R_PPC64_TOC16	47	half16*	S + A - .TOC.
R_PPC64_TOC16_LO	48	half16	#lo(S + A - .TOC.)
R_PPC64_TOC16_HI	49	half16	#hi(S + A - .TOC.)
R_PPC64_TOC16_HA	50	half16	#ha(S + A - .TOC.)
R_PPC64_TOC	51	doubleword64	.TOC.
R_PPC64_PLTGOT16	52	half16*	M
R_PPC64_PLTGOT16_LO	53	half16	#lo(M)
R_PPC64_PLTGOT16_HI	54	half16	#hi(M)
R_PPC64_PLTGOT16_HA	55	half16	#ha(M)
R_PPC64_ADDR16_DS	56	half16ds*	(S + A) >> 2
R_PPC64_ADDR16_LO_DS	57	half16ds	#lo(S + A) >> 2
R_PPC64_GOT16_DS	58	half16ds*	G >> 2
R_PPC64_GOT16_LO_DS	59	half16ds	#lo(G) >> 2
R_PPC64_PLT16_LO_DS	60	half16ds	#lo(L) >> 2
R_PPC64_SECTOFF_DS	61	half16ds*	(R + A) >> 2
R_PPC64_SECTOFF_LO_DS	62	half16ds	#lo(R + A) >> 2
R_PPC64_TOC16_DS	63	half16ds*	(S + A - .TOC.) >> 2
R_PPC64_TOC16_LO_DS	64	half16ds	#lo(S + A - .TOC.) >> 2
R_PPC64_PLTGOT16_DS	65	half16ds*	M >> 2
R_PPC64_PLTGOT16_LO_DS	66	half16ds	#lo(M) >> 2
R_PPC64_TLS	67	none	none
R_PPC64_DTPMOD64	68	doubleword64	@dtpmod
R_PPC64_TPREL16	69	half16*	@tprel
R_PPC64_TPREL16_LO	60	half16	#lo(@tprel)
R_PPC64_TPREL16_HI	71	half16	#hi(@tprel)
R_PPC64_TPREL16_HA	72	half16	#ha(@tprel)
R_PPC64_TPREL64	73	doubleword64	@tprel
R_PPC64_DTPREL16	74	half16*	@dtprel
R_PPC64_DTPREL16_LO	75	half16	#lo(@dtprel)
R_PPC64_DTPREL16_HI	76	half16	#hi(@dtprel)
R_PPC64_DTPREL16_HA	77	half16	#ha(@dtprel)
R_PPC64_DTPREL64	78	doubleword64	@dtprel
R_PPC64_GOT_TLSGD16	79	half16*	@got@tlsgd
R_PPC64_GOT_TLSGD16_LO	80	half16	#lo(@got@tlsgd)
R_PPC64_GOT_TLSGD16_HI	81	half16	#hi(@got@tlsgd)
R_PPC64_GOT_TLSGD16_HA	82	half16	#ha(@got@tlsgd)
R_PPC64_GOT_TLSLD16	83	half16*	@got@tlsld
R_PPC64_GOT_TLSLD16_LO	84	half16	#lo(@got@tlsld)
R_PPC64_GOT_TLSLD16_HI	85	half16	#hi(@got@tlsld)
R_PPC64_GOT_TLSLD16_HA	86	half16	#ha(@got@tlsld)

Name[1]	Value[2]	Field[3]	Calculation[4]
R_PPC64_GOT_TPREL16_DS	87	half16ds*	@got@tprel
R_PPC64_GOT_TPREL16_LO_DS	88	half16ds	#lo(@got@tprel)
R_PPC64_GOT_TPREL16_HI	89	half16	#hi(@got@tprel)
R_PPC64_GOT_TPREL16_HA	90	half16	#ha(@got@tprel)
R_PPC64_GOT_DTPREL16_DS	91	half16ds*	@got@dtprel
R_PPC64_GOT_DTPREL16_LO_DS	92	half16ds	#lo(@got@dtprel)
R_PPC64_GOT_DTPREL16_HI	93	half16	#hi(@got@dtprel)
R_PPC64_GOT_DTPREL16_HA	94	half16	#ha(@got@dtprel)
R_PPC64_TPREL16_DS	95	half16ds*	@tprel
R_PPC64_TPREL16_LO_DS	96	half16ds	#lo(@tprel)
R_PPC64_TPREL16_HIGHER	97	half16	#higher(@tprel)
R_PPC64_TPREL16_HIGHERA	98	half16	#highera(@tprel)
R_PPC64_TPREL16_HIGHEST	99	half16	#highest(@tprel)
R_PPC64_TPREL16_HIGHESTA	100	half16	#highesta(@tprel)
R_PPC64_DTPREL16_DS	101	half16ds*	@dtprel
R_PPC64_DTPREL16_LO_DS	102	half16ds	#lo(@dtprel)
R_PPC64_DTPREL16_HIGHER	103	half16	#higher(@dtprel)
R_PPC64_DTPREL16_HIGHERA	104	half16	#highera(@dtprel)
R_PPC64_DTPREL16_HIGHEST	105	half16	#highest(@dtprel)
R_PPC64_DTPREL16_HIGHESTA	106	half16	#highesta(@dtprel)
R_PPC64_TOC32	107	word32	(.TOC. & 0xffff_ffff)
R_PPC64_DTPMOD32	108	word32	(@dtpmod & 0xffff_ffff)
R_PPC64_TPREL32	109	word32	(@tprel & 0xffff_ffff)
R_PPC64_DTPREL32	110	word32	(@dtprel & 0xffff_ffff)

[1] The relocation types whose names include `_BRTAKEN` or `_BRNTAKEN` specify whether the branch prediction bit (bit 10) should indicate that the branch will be taken or not taken, respectively. For an unconditional branch, the branch prediction bit must be 0.

[2] Relocation values 18, 23 and 32 are not used. Relocations 56-66 are used for instructions with a DS offset field (`ld`, `ldu`, `lwa`, `std`, `stdu`). ABI-conformant tools should return an error if there is an attempt to relocate an address to a value that is not divisible by 4.

[3] The relocation types whose Field column entry contains an asterisk are subject to failure if the value computed does not fit in the allocated bits.

[4] The following notation is used to describe the Calculation entry in the table:

- The letters *A*, *B*, *G*, *L*, *M*, *P*, *R*, and *S*, and the term *.TOC.* represent:
  - *A*: The addend used to compute the value of the relocatable field.
  - *B*: The base address at which a shared object has been loaded into memory during execution. Generally, a shared object file is built with a 0 base virtual address, but the execution address will be different. See the “Program Header” section in the System V ABI for more information about the base address.
  - *G*: The offset into the global offset table, relative to the TOC base, at which the address of the relocation entry’s symbol plus addend will reside during execution. A reference in a calculation to the value G implicitly creates a GOT entry for the indicated symbol.
    For more information, see “Coding Examples” and “Global Offset Table”.
  - *L*: Relocation types containing this parameter are not used.
  - *M*: Relocation types containing this parameter are not used.
  - *P*: The place (section offset or address) of the storage unit being relocated.
    This is computed using *r_offset*.
  - *R*: The offset of the symbol within the section in which the symbol is defined (its section-relative address).
  - *S*: The value of the symbol whose index is located in the relocation entry.

  - `.TOC.`: The TOC base of the TOC section for the object being relocated. See "TOC" for additional information. The dynamic linker does not have this information, and thus relocation types that refer to `.TOC.` may only appear in relocatable object files, not in executables or shared objects.
- "+" and "-" denote 64-bit modulus addition and subtraction, respectively. ">>" denotes arithmetic right-shifting (shifting with sign copying) of the value of the left operand by the number of bits given by the right operand.
- For relocation types whose name contains "30", the upper 33 bits of the value computed before shifting must be the same.
- For relocation types whose name contains "14" or "16", the upper 49 bits of the value computed before shifting must be the same.
- For relocation types whose name contains "24", the upper 39 bits of the value computed before shifting must be the same.
- For relocation types whose name contains "14" or "24", the low 2 bits of the value computed before shifting must be zero.
- The notations that begin with "`@`" denote relocations that are used with thread-local symbols; these notations have the indicated meanings:
  - *`@dtpmod`*: Computes the load module index of the load module that contains the definition of sym. The addend, if present, is ignored.
  - *`@dtprel`*: Computes a dtp-relative displacement. This displacement is the difference between the value of `S` + `A` and the base address of the thread-local storage block that contains the definition of the symbol, minus `0x8000`.
  - *`@tprel`*: Computes a tp-relative displacement. This displacement is the difference between the value of `S` + `A` and the value of the thread pointer (`r13`).
  - *`@got@tlsgd`*: Allocates two contiguous entries in the GOT to hold a `tls_index` structure, with values `@dtpmod` and `@dtprel`, and computes the offset to the first entry relative to the TOC base (`r2`).
  - *`@got@tlsld`*: Allocates two contiguous entries in the GOT to hold a `tls_index` structure, with values `@dtpmod` and zero, and computes the offset to the first entry relative to the TOC base (`r2`).
  - *`@got@dtprel`*: Allocates an entry in the GOT with value `@dtprel`, and computes the offset to the entry relative to the TOC base (`r2`).
  - *`@got@tprel`*: Allocates an entry in the GOT with value `@tprel`, and computes the offset to the entry relative to the TOC base (`r2`).
- `#lo(`*`value`*`)` denotes the least significant 16 bits of the indicated value:
  `#lo(x) = (x & 0xffff).`
- `#hi(`*`value`*`)` denotes bits 16 through 31 of the indicated value:
  `#hi(x) = ((x >> 16) & 0xffff).`
- `#ha(`*`value`*`)` denotes the high adjusted value: bits 16 through 31 of the indicated value, compensating for `#lo()` being treated as a signed number:
  `#ha(x) = (((x >> 16) + ((x & 0x8000) ? 1 : 0)) & 0xffff)`
- `#higher(`*`value`*`)` denotes bits 32 through 47 of the indicated value:
  `#higher(x) = ((x >> 32) & 0xffff)`
- `#highera(`*`value`*`)` denotes bits 32 through 47 of the indicated value, compensating for `#lo()` being treated as a signed number:
  `#highera(x) = (((x >> 32) + (((x & 0xffff8000) == 0xffff8000) ? 1 : 0)) & 0xffff)`
- `#highest(`*`value`*`)` denotes bits 48 through 63 of the indicated value:
  `#highest(x) = ((x >> 48) & 0xffff)`
- `#highesta(`*`value`*`)` denotes bits 48 through 63 of the indicated value, compensating for `#lo()` being treated as a signed number:
  `#highesta(value) = (((x >> 48) + (((x & 0xffffffff8000) == 0xffffffff8000) ? 1 : 0)) & 0xffff)`

Relocation types with special semantics are described in Table 13.

Table 13 Relocation Types with Special Semantics

Name	Description
R_PPC64_GOT16*	These relocation types resemble the corresponding R_PPC64_ADDR16* types, except that they refer to the address of the symbol's global offset table entry and additionally instruct the link editor to build a global offset table.
R_PPC64_PLTGOT16*	These relocation types are not used.
R_PPC64_COPY	The link editor creates this relocation type for dynamic linking. Its offset member refers to a location in a writable segment. The symbol table index specifies a symbol that should exist both in the current object file and in a shared object. During execution, the dynamic linker copies data associated with the shared object's symbol to the location specified by the offset.
R_PPC64_GLOB_DAT	This relocation type resembles R_PPC64_ADDR64, except that it sets a global offset table entry to the address of the specified symbol. This special relocation type allows one to determine the correspondence between symbols and global offset table entries.
R_PPC64_JMP_SLOT	This relocation type is not used.
R_PPC64_RELATIVE	The link editor creates this relocation type for dynamic linking. Its offset member gives a location within a shared object that contains a value representing a relative address. The dynamic linker computes the corresponding virtual address by adding the virtual address at which the shared object was loaded to the relative address. Relocation entries for this type must specify 0 for the symbol table index.
R_PPC64_UADDR*	These relocation types are the same as the corresponding R_PPC64_ADDR* types, except that the datum to be relocated is allowed to be unaligned.

Program Loading and Dynamic Linking

Program Loading

As the system creates or augments a process image, it logically copies a file's segment to a virtual memory segment. When or whether the system physically reads the file depends on the program's execution behavior, system load, and so forth. A process does not require a physical page unless it references the logical page during execution, but processes commonly leave many pages unreferenced. Therefore, delaying physical reads frequently means that a read never occurs; this improves system performance. To obtain this efficiency in practice, executable and shared object files must have segment images whose offsets and virtual addresses are congruent, modulo the page size.

Virtual addresses and file offsets for PPU segments are congruent modulo 64 kilobytes (`0x10000`) or larger powers of 2. The Cell Broadband Engine™ Architecture defines multiple page sizes; therefore, to perform suitable paging, the *`p_align`* member of each program header in a shared object file must be `0x10000`.

It is normally desirable to put segments with different characteristics in separate 256 megabyte portions of the address space, to give the operating system full paging flexibility in the 32-bit address space.

The following is an example of an executable file assuming an executable program linked with a base address of `0x10000`.

```
File Offset                             Virtual Address

0x0000_0000
     Text segment
            ELF header                  0x0001_0000
            Program header table
            Other information
            .init                       0x0001_0120
            .text
            .fini
            other read-only sections
            0xe4a0 bytes
                                        0x0001_e4a0
0x0000_f000
      Data Segment
            .data                       0x0002_f000
            .bss
            other read-write sections
            0x1f10 bytes
                                        0x0003_0f10
```

Table 14 shows possible corresponding program header segments.

Table 14 Possible Corresponding Program Header Segments

Member	Text	Data
p_type	`PT_LOAD`	`PT_LOAD`
p_offset	`0x0`	`0xf000`
p_vaddr	`0x0001_0000`	`0x0002_f000`
p_paddr	unspecified	unspecified
p_filesz	`0xe4a0`	`0xd74`
p_memsz	`0xe4a0`	`0x1f10`
p_flags	`PF_R+PF_X`	`PF_R+PF_W`
p_align	`0x10000`	`0x10000`

The example addresses for the text and data segments are suggested but are not required.

Although the file offsets and virtual addresses are congruent modulo 64 kilobytes for both text and data, up to four file pages can hold impure text or data (depending on page size and file system block size):

- The first text page contains the ELF header, the program header table, and other information.
- The last text page may hold a copy of the beginning of data.
- The first data page may have a copy of the end of text.
- The last data page may contain file information not relevant to the running process.

Logically, the system enforces memory permissions as if each segment were complete and separate; segment addresses are adjusted to ensure that each logical page in the address space has a single set of permissions. In the example above, the file region holding the end of text and the beginning of data is mapped twice: at one virtual address for text and at a different virtual address for data.

The end of the data segment requires special handling for uninitialized data, which the system defines to begin with zero values. Thus, if the last data page of a file includes information not in the logical memory page, the extraneous data must be set to zero, rather than to the unknown contents of the executable file. "Impurities" in the other three pages are not logically part of the process image; whether the system eliminates them is unspecified. Figure 21 shows the memory image for the program above, assuming 4096 (`0x1000`) byte pages.

Figure 21 Virtual Address

```
Text segment:
0x0001_0000  Header and padding  0x120  bytes
0x0001_0120  Text segment        0xe380 bytes
             Padding             0xb60  bytes
Data Segment:
0x0001_f000  Initialized data    0xd74  bytes
0x0001_fd74  Uninitialized data  0x119c bytes
             Padding             0xf0   bytes
```

One aspect of segment loading differs between executable files and shared objects: Executable file segments may contain absolute code. For the process to execute correctly, the segments must reside at the virtual addresses assigned when building the executable file, with the system using the *p_vaddr* values unchanged as virtual addresses.

On the other hand, shared object segments typically contain position-independent code. This allows a segment's virtual address to change from one process to another, without invalidating execution behavior.

Although the system chooses virtual addresses for individual processes, most systems will maintain the "relative positions" of the segments. Any use of relative addressing between segments should be indicated by an appropriate dynamic relocation. If the dynamic linker does not maintain the relative position of segments at load time, it must be careful in its handling of `R_PPC64_RELATIVE` relocations, examining the relative address in order to determine the appropriate base address.

Table 15 shows possible shared object virtual address assignments for several processes, illustrating constant relative positioning. The table also illustrates the base address computations.

Table 15 Shared Object Virtual Address Assignments

Source	Text	Data	Base Address
File	0x000200	0x02a400	
Process 1	0x100200	0x12a400	0x100000
Process 2	0x200200	0x22a400	0x200000
Process 3	0x300200	0x32a400	0x300000
Process 4	0x400200	0x42a400	0x400000

Program Interpreter

The Cell OS Lv-2 does not support a program interpreter, such as `ld.so.1`. Therefore, interpreter information is always ignored.

Dynamic Linking

Dynamic Section

Dynamic section entries give information to the dynamic linker; some of this information is processor-specific, including the interpretation of some entries in the dynamic structure.

Global Offset Table

Position-independent code cannot, in general, contain absolute virtual addresses. The global offset table, which is part of the TOC section, holds absolute addresses in private data, thus making the addresses available without compromising the position-independence and sharability of a program's text. A program references its TOC using position-independent addressing and extracts absolute values, thus redirecting position-independent references to absolute locations.

When the dynamic linker creates memory segments for a loadable object file, it processes the relocation entries, some of which will be of type `R_PPC64_GLOB_DAT`, referring to the global offset table within the TOC. The dynamic linker determines the associated symbol values, calculates their absolute addresses, and sets the global offset table entries to the proper values. Although the absolute addresses are unknown

when the link editor builds an object file, the dynamic linker knows the addresses of all memory segments and can thus calculate the absolute addresses of the symbols contained therein.

A global offset table entry provides direct access to the absolute address of a symbol without compromising position-independence and sharability. Because the executable file and shared objects have separate global offset tables, a symbol may appear in several tables. The dynamic linker processes all the global offset table relocations before giving control to any code in the process image, thus ensuring that the absolute addresses are available during execution.

The global offset table is part of the TOC section. Because different functions in a single executable or shared object may have different TOC sections, the global offset table may also be replicated, in whole or in part. Each instance of the global offset table will have its own set of relocations. The dynamic linker does not need to know about the replication; it simply processes all the relocations it is given.

The dynamic linker may choose different memory segment addresses for the same shared object in different programs; it may even choose different library addresses for different executions of the same program. Nevertheless, memory segments do not change addresses after the process image is created. As long as a process exists, its memory segments reside at fixed virtual addresses.

The global offset table normally resides in the ELF .got section in an executable or shared object.

Function Addresses

References to the address of a function from an executable file and the shared objects associated with it need to resolve to the same value.

In this ABI, the address of a function is actually the address of a function descriptor. A reference to a function, other than a function call, will normally load the address of the function descriptor from the global offset table. The dynamic linker will ensure that for a given function, the same address is used for all references to the function from any global offset table. Thus, function address comparisons will work as expected.

When making a call to the function, the code may refer to the procedure linkage table entry, in order to permit lazy symbol resolution at runtime. In order to support correct function address comparisons, the compiler should be careful to only generate references to the procedure linkage table entry for function calls. For any other use of a function, the compiler should use the real address.

When using the ELF assembler syntax, this means that the compiler should use the `@got` syntax, rather than the `@got@plt` syntax, if the function address is going to be used without being called.

Libraries

This document does not specify any library interfaces.

APPENDIX A: Primary Differences Between PPU ABI for Cell OS Lv-2 and 64-Bit PowerPC ELF ABI

This appendix identifies the primary differences between the PPU ABI for Cell OS Lv-2 and the 64-bit PowerPC ELF ABI.

- The size and alignment of a pointer are each 4 bytes. The PowerPC ABI defines these as 8 bytes.
- The size and alignment of `long` are each 4 bytes. The PowerPC ABI defines these as 8 bytes.
- The "plain" `char` type is `signed char` whereas the plain `char` type is an `unsigned char` in the PowerPC ABI.
- Additional data types have been defined to represent some fundamental data types for OS and library interfaces. (See Table 2.)
- Each function descriptor has two 32-bit address fields and no environment pointer. As a result, this ABI does not support nested functions. Nested functions are a GCC language extension. For the PowerPC ABI, each function descriptor consists of three 64-bit address fields: the address of the function body, a pointer to the TOC referred to by the function, and the environment pointer used by the function.
- The initial values of some registers have been changed, for process initialization.
- New relocation types, `R_PPC64_TOC32`, `R_PPC64_DTPMOD32`, `R_PPC64_TPREL32` and `R_PPC64_DPTPREL32` are defined.
- The OS and ABI identifier in the ELF file header is defined as 102 for Cell OS Lv-2.
- `VRSAVE` is not required to be updated at runtime; however, it should be initialized to `0xffff_ffff`.
- Traceback tables are not supported.
- The "extended precision" format is not supported.